APT Bahamut Attacks Indian Intelligence Operative using Android Malware - CYFIRMA

Archived: 2026-04-05 14:55:48 UTC

Published On : 2023-02-10

Executive Summary

In November 2022, CYFIRMA detected a cyber-attack on an intelligence operative in India. In this attack, the threat actor was seen leveraging a strategic social engineering attack to deliver and install the .APK on the victim's mobile. The threat actor requested the victim to share unknown files in encrypted form via an android app, which was malicious. The malicious app was instantly attached with a direct message on telegram. Upon installing, the malicious Android Package, an app with a random icon led to a dummy sign-up and login page, followed by setting up a new pattern lock. The icon after installation and the icon after opening the app were both different from each other. The threat actor has created the app as a decoy of an encrypted file-sharing app. However, the malicious .APK is just a dummy app with a pattern lock, login page, and sign-up page. The layer of the pattern lock page is inserted to make the victim believe the app is a genuine encrypted file-sharing platform.

ETLM Attribution

Research revealed the threat actor Bahamut was behind the attack. Our team with high confidence attributed the attack to Bahamut APT. Through our investigation, we have found that IOCs that we fetched from the malicious android package were earlier associated with the Bahamut. The threat actor is known for conducting cyber strikes in the middle eastern region and South Asian region. The code in previous malicious apps that was used by Bahamut is similar to the .APK that was used in the recent attack. Earlier .APK with the name “SecureVpn” (b65a8edc06bbeb598e495ccc44dc40e77ab2ef0ab11e136a0a10c24970640b42) was widely used for mass attacks on Android users. However, this is the first time, we have observed Bahamut using a fake Secure File sharing app in a strategic social engineering attack. This is the first time CYFIRMA observed the threat actor Bahamut targeting intelligence operatives, in fact, cyber-attack on intelligent assets is a rare occurrence in cyber espionage operations. Bahamut is specialized in targeting individuals with strategic social engineering attacks, they take time and showcase the ultimate level of patience while engaging with victims.

The threat actor leveraged strategic social engineering. CYFIRMA was able to get the screenshot of actual the attack taking place through telegram. In the below screenshot, the threat actor asks the victim to use the app to share the files in encrypted form. The threat actor kept the engagement going for the past few days and suddenly took leverage of earned trust to make the victim install the app. However, the attack was detected and dismantled before it could leave any damage.

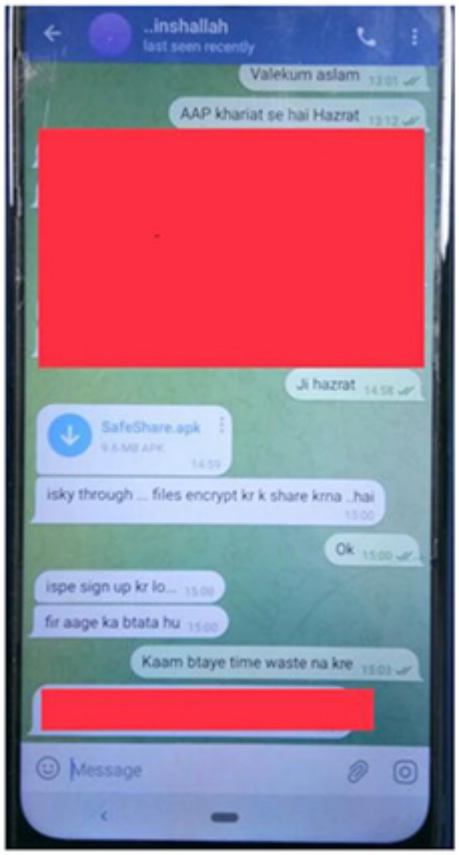


Figure 1. Screenshots of Bahamut delivering .APK payload.

Analysis of the Sample

File: SafeShare.apk
System: Android System

MD5: 76b6ff206d11cadde52f37df0d19eecd
SHA1: 34538c92c5e48b62f6523bcbe9961d592a41e32d
SHA256: 45a6a0b2b02a9d288afba1ff41c689be9b9bd40ee862aa4bd6b036e3f0a4c3ab Package Name: com.secure.vault
File Type: APK

Process Overview

Upon Installation, an app with the name Vault reflects on Apps Menu.

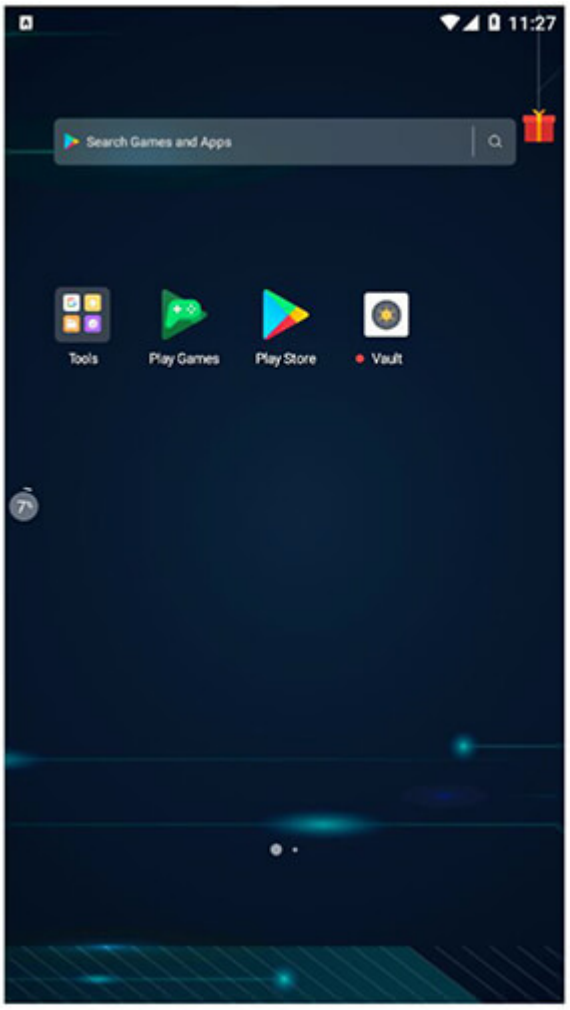


Figure 2. Vault application stationed in App Menu.

The below figure shows that the Android Vault Application requests access permission at the start.

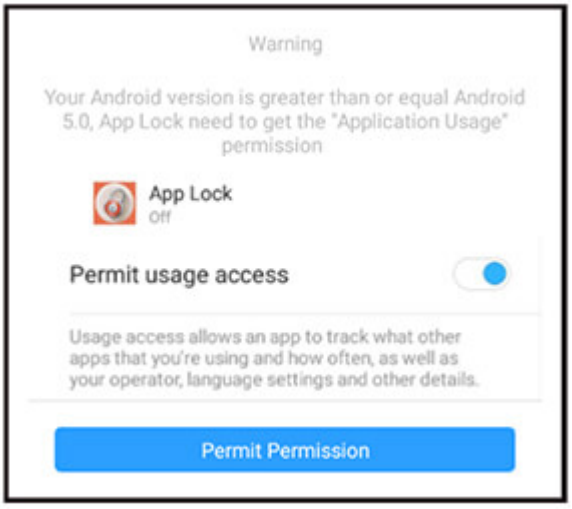


Figure 3. Usage access permission.

After opening the app, it takes the victim to set the pattern lock. This page was added to make the victim believe that the app is a legitimate secure file-sharing app.

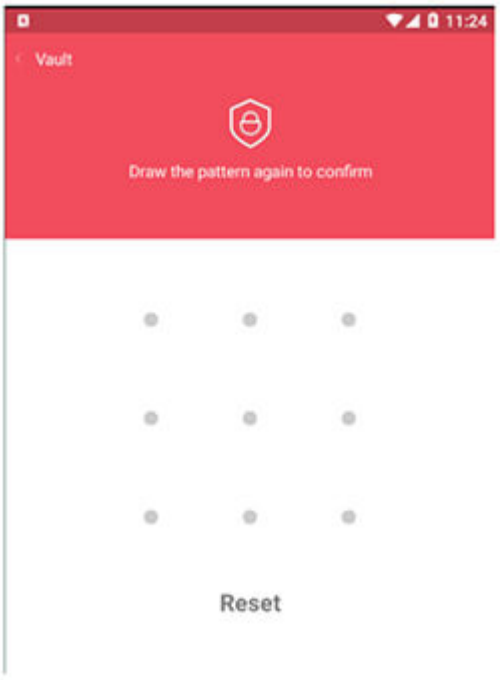


Figure 4. Setup Pattern Lock

After setting the pattern lock, the app opens up the next page with login and Sign-Up options.

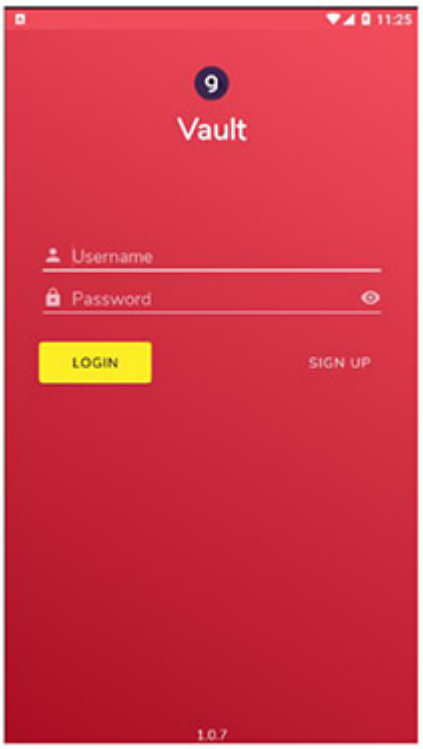


Figure 5. Login Activity.

This is the last page, where the victim is supposed to register for the service.

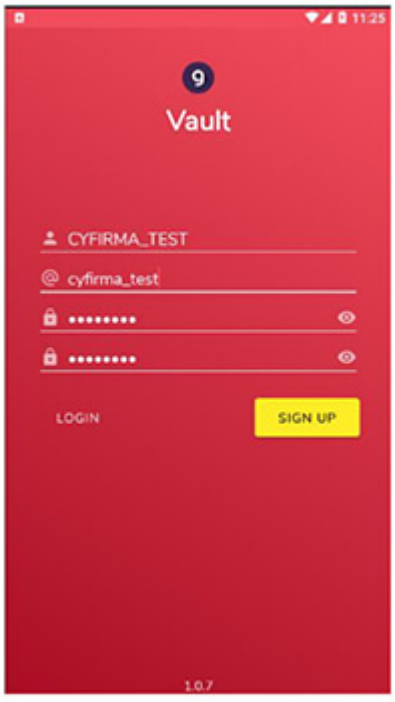


Figure 6. Dummy Sign-Up Page.

Code Overview

Application Certificates and Signatures

The META-INF folder contains the signature file and self-signed public key certificate. GOOGLE.RSA certificate

can be viewed using the key tool.

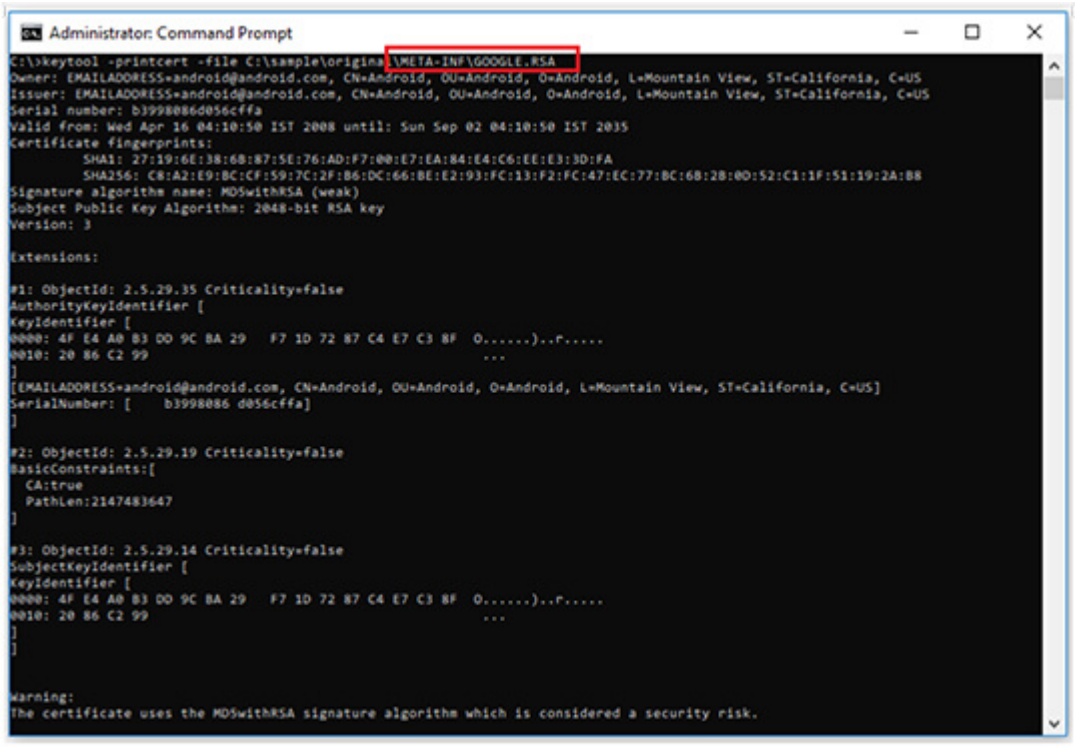


Figure 7. Keytool.

AndroidManifest File

This XML contains the necessary essential information about the application and its components, required permissions, used libraries and Java packages, and more, to run on the Android System.

As the manifest file is encoded, decode the file contents using the apktool with the d option.

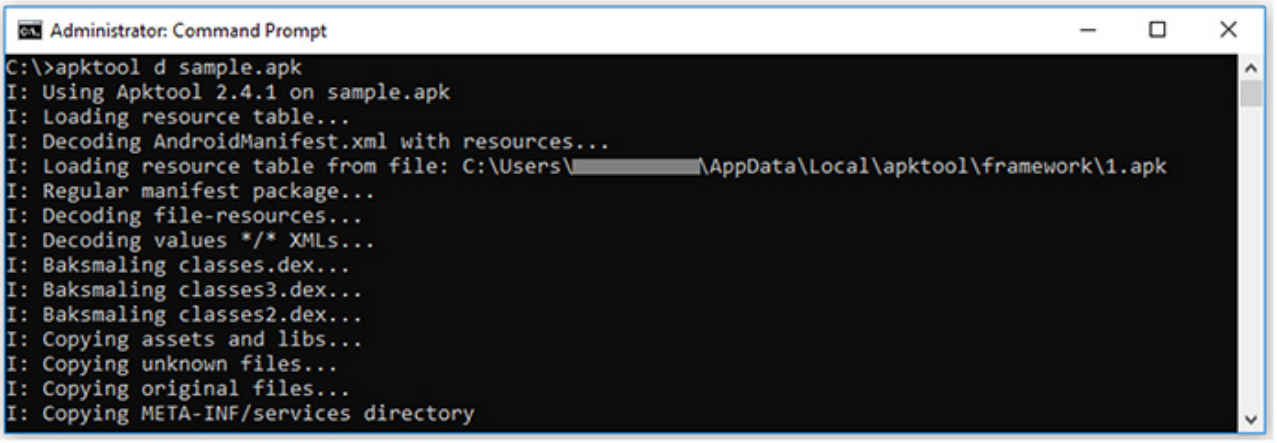


Figure 8. Apktool d option.

The apktool decodes all binary XML files, including Android manifests and resources, but also disassembles the DEX file with baksmali. The apktool generates, [.]smali files, which can be examined by any text editor.

The apktool output shows the below files/folders structure.

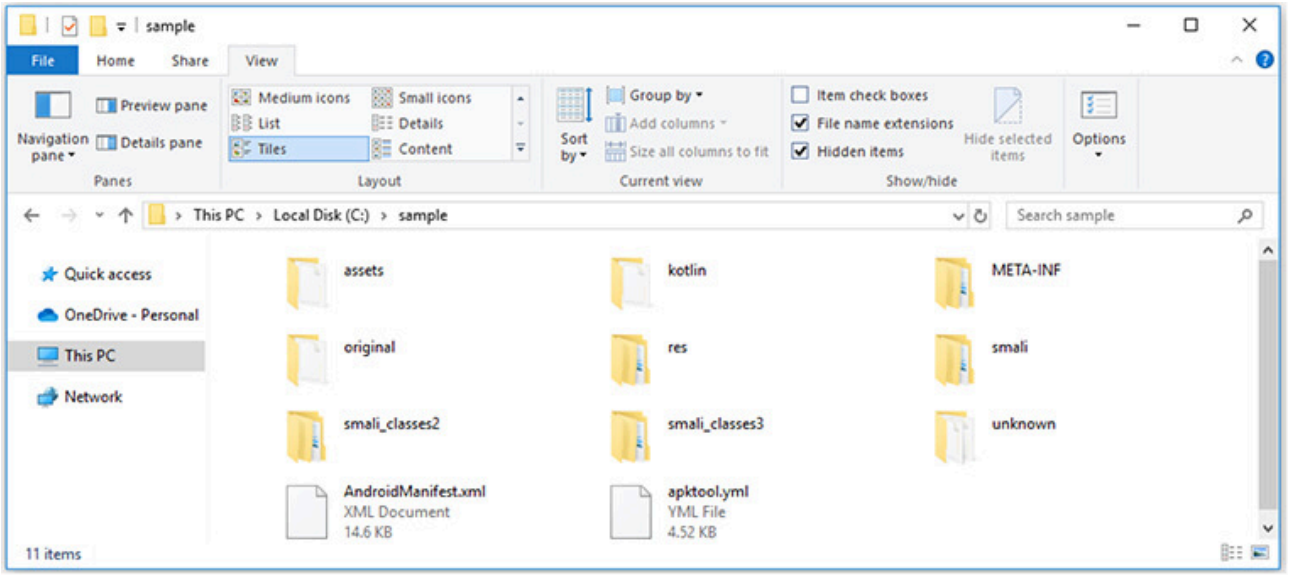


Figure 9. APKtool Output.

Application main activities (activities are nothing but pages in the App) & icon information from the manifest file.

Figure 10. Main Activity & Icon.

SDK & API Level information from the manifest file.

Figure 11. SDK & API Level information.

Permissions from the manifest file. The permissions clearly indicate the malicious .APK was put on the job to fetch maximum information from the victim's mobile.

```
<uses-permission android:name="android.permission.READ_CALL_LOG"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.USE_BIOMETRIC"/>
<uses-permission android:name="android.permission.USE_FINGERPRINT"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.CALL_PHONE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
<uses-permission android:name="android.permission.PACKAGE_USAGE_STATS"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
<uses-feature android:name="android.hardware.camera"/>
<uses-permission android:maxSdkVersion="22" android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

Figure 12. Permissions.

Below are a few dangerous permissions that are malicious. .APK is gaining access to them.

Sr.no	Tactics	Technique ID
1	READ_SMS	This allows attackers to delete and read outgoing and incoming SMSs
2	READ_CALL_LOG	This allows actors to read and fetch call logs.
3	READ_CONTACTS	This permission allows TA to read and fetch contacts.
4	READ_EXTERNAL_STORAGE	This allows threat actors to explore and fetch data from the file manager.
5	WRITE_EXTERNAL_STORAGE	This allows threat actors to delete and move files.
6	GET_ACCOUNTS	This allows the threat actor to extract emails and usernames used for login into various internet platforms
7	CAMERA	This allows the threat actor to use a front and back camera.
8	ACCESS_FINE_LOCATION	Allows the threat actor to fetch precise locations and track the live movement of mobile phones.
9	WRITE_CALL_LOG	This allows the threat actor to make calls and delete them from call logs.
10	WRITE_CONTACTS	This allows the threat actor to delete and add contacts.

Android Vault Application collects information on the device’s contacts, SMS, and call logs in the initial stage.

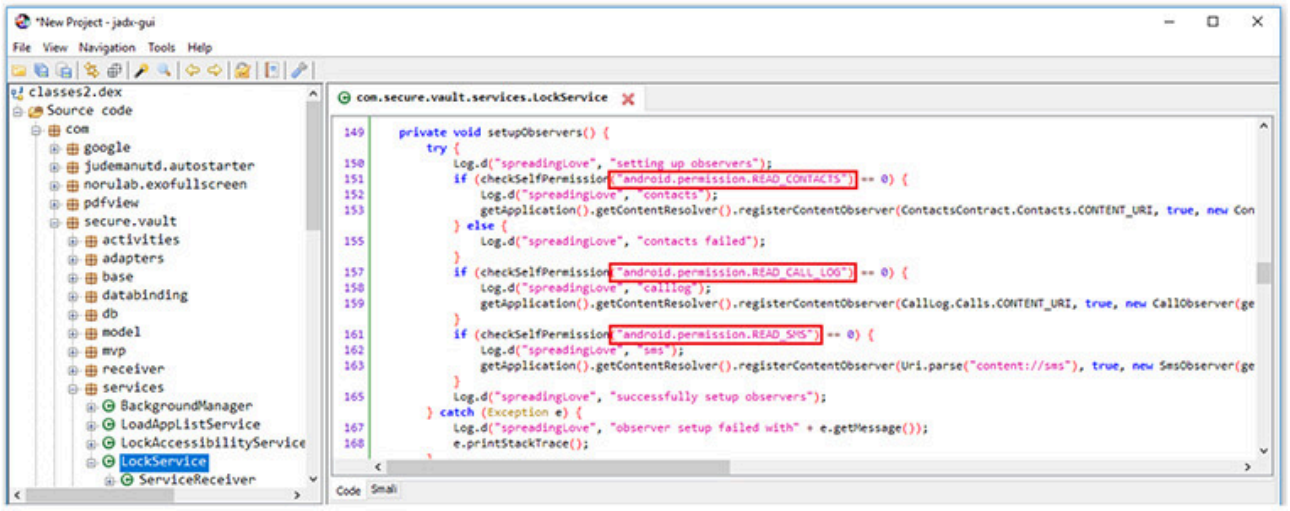


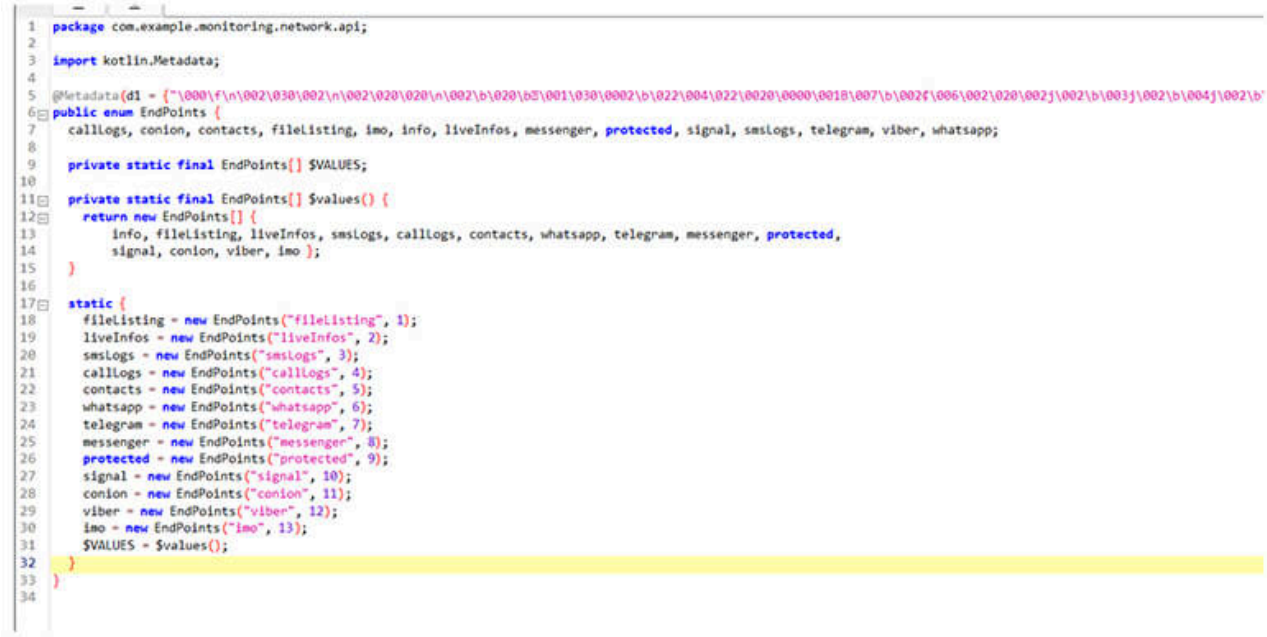
Figure 13. Lock service for collecting information.

Below is the module, which exploits “android.permission.ACCESS_FINE_LOCATION” permission to fetch the precise location of the hacked user.

```
private Location getLastKnownLocation() {
  Location location1;
  int i = PermissionChecker.checkSelfPermission(this.mContext, "android.permission.ACCESS_COARSE_LOCATION");
  Location location2 = null;
  if (i == 0) {
    location1 = getLastKnownLocationForProvider("network");
  } else {
    location1 = null;
  }
  if (PermissionChecker.checkSelfPermission(this.mContext, "android.permission.ACCESS_FINE_LOCATION") == 0)
    location2 = getLastKnownLocationForProvider("gps");
  if (location2 != null && location1 != null) {
    Location location = location1;
    if (location2.getTime() > location1.getTime())
      location = location2;
    return location;
  }
  if (location2 != null)
    location1 = location2;
  return location1;
}
```

Figure 14. Exploiting one of the permissions.

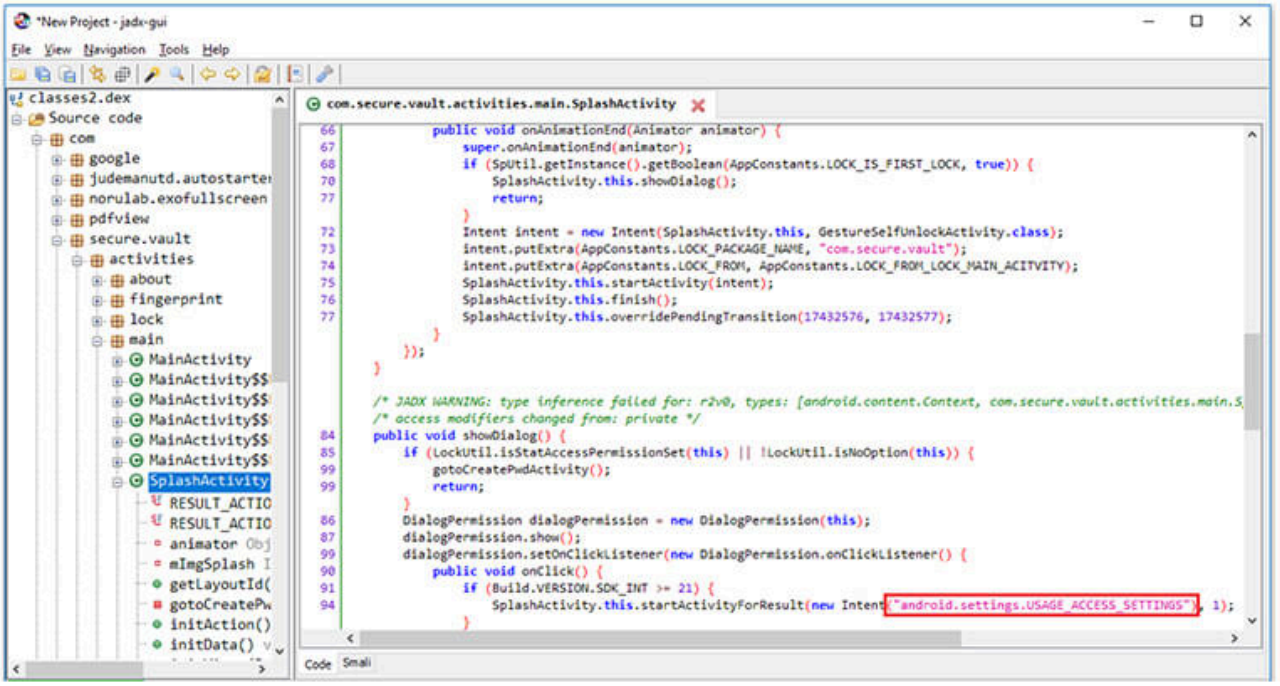
Below is the module, revealing what sort of information this malicious .APK collects, and sends to the C2. It fetches call logs, SMS logs, and live coordinates and tracks messages by keystrokes, by abusing accessibility from apps like Telegram, Signal, Viber, IMO, and Conion.



```
package com.example.monitoring.network.api;

import kotlin.Metadata;

@Metadata(d1 = {"\000\f\n\002\030\002\n\002\020\020\n\002\b\020\b\001\030\0002\b\022\004\022\0020\0000\001B\007\b\002\006\002\020\002j\002\b\003j\002\b\004j\002\b
public enum EndPoints {
  callLogs, conion, contacts, fileListing, imo, info, liveInfos, messenger, protected, signal, smsLogs, telegram, viber, whatsapp;

  private static final EndPoints[] $VALUES;

  private static final EndPoints[] $values() {
    return new EndPoints[] {
        info, fileListing, liveInfos, smsLogs, callLogs, contacts, whatsapp, telegram, messenger, protected,
        signal, conion, viber, imo };
  }

  static {
    fileListing = new EndPoints("fileListing", 1);
    liveInfos = new EndPoints("liveInfos", 2);
    smsLogs = new EndPoints("smsLogs", 3);
    callLogs = new EndPoints("callLogs", 4);
    contacts = new EndPoints("contacts", 5);
    whatsapp = new EndPoints("whatsapp", 6);
    telegram = new EndPoints("telegram", 7);
    messenger = new EndPoints("messenger", 8);
    protected = new EndPoints("protected", 9);
    signal = new EndPoints("signal", 10);
    conion = new EndPoints("conion", 11);
    viber = new EndPoints("viber", 12);
    imo = new EndPoints("imo", 13);
    $VALUES = $values();
  }
}
```

Figure 15. List of Information, the malicious app steals from compromised users.

Analyze the code from SplashActivity, as the app execution starts from the main activity & Application subclasses.




```
            public void onAnimationEnd(Animator animator) {
                super.onAnimationEnd(animator);
                if (SpUtil.getInstance().getBoolean(AppConstants.LOCK_IS_FIRST_LOCK, true)) {
                    SplashActivity.this.showDialog();
                    return;
                }
                Intent intent = new Intent(SplashActivity.this, GestureSelfUnlockActivity.class);
                intent.putExtra(AppConstants.LOCK_PACKAGE_NAME, "com.secure.vault");
                intent.putExtra(AppConstants.LOCK_FROM, AppConstants.LOCK_FROM_LOCK_MAIN_ACITVITY);
                SplashActivity.this.startActivity(intent);
                SplashActivity.this.finish();
                SplashActivity.this.overridePendingTransition(17432576, 17432577);
            }
        });
    }

    /* JADX WARNING: type inference failed for: r2v0, types: [android.content.Context, com.secure.vault.activities.main.S
    /* access modifiers changed from: private */
    public void showDialog() {
        if (LockUtil.isStatAccessPermissionSet(this) || !LockUtil.isNoOption(this)) {
            gotoCreatePwdActivity();
            return;
        }
        DialogPermission dialogPermission = new DialogPermission(this);
        dialogPermission.show();
        dialogPermission.setOnClickListener(new DialogPermission.onClickListener() {
            public void onClick() {
                if (Build.VERSION.SDK_INT >= 21) {
                    SplashActivity.this.startActivityForResult(new Intent("android.settings.USAGE_ACCESS_SETTINGS"), 1);
                }
```

Figure 16. Usage Access Setting

Android API

Android APIs used in this sample is based on static analysis.

Get SIM Serial Number
Inter-Process Communication
Get System Service

URL Connection
Base64 Encode
Starting Service
UDP Datagram Socket
Android Notifications
Query Database of SMS, Contacts, etc. TCP Socket
Message Digest
Get Device ID, IMEI, MEID/ESN etc. Crypto
Base64 Decode
Starting Activity
Java Reflection Method Invocation
URL Connection to file/http/https/ftp/jar Local File I/O Operations
HTTP Connection
GPS Location

The const-string is being used to move a value (string value) into the registers (v0, v1, etc.). The invoke- direct is used to invoke the instance methods. The invoke-direct accepts two arguments (po register and a reference to the method that needs to be called).

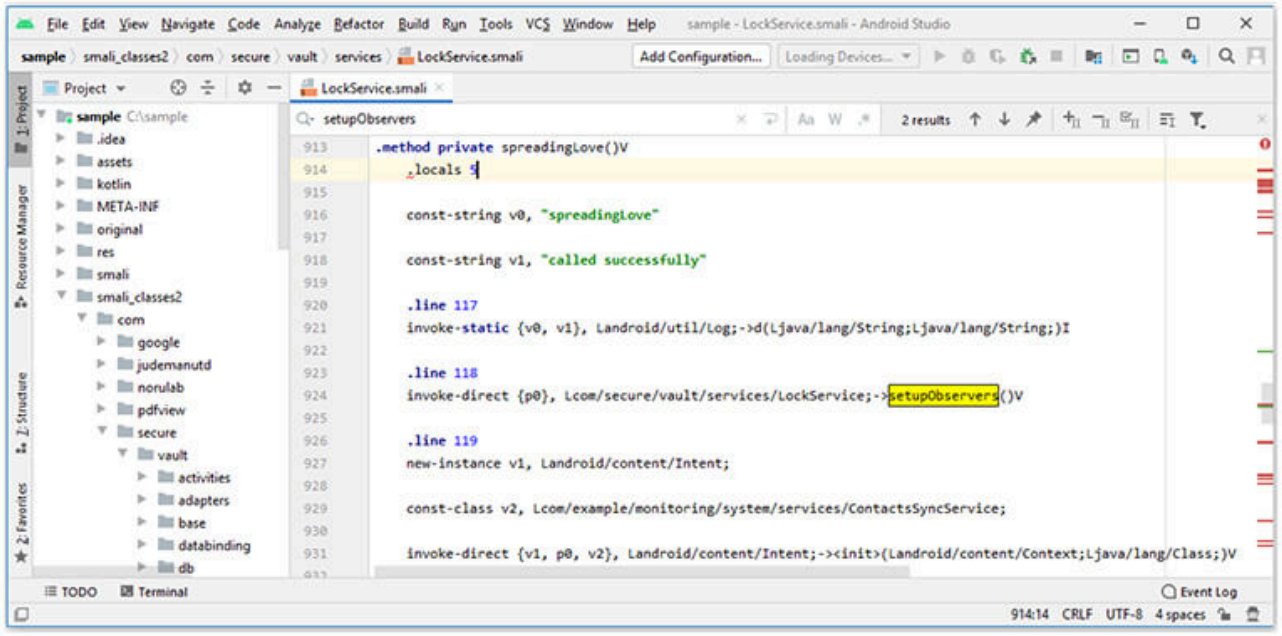


Figure 17. Smali Code Analysis.

The below screenshot shows the module collects data in SQL queries.

Figure 18. SQL queries.

The module contains DNS and IP which belongs to the command-and-control server.

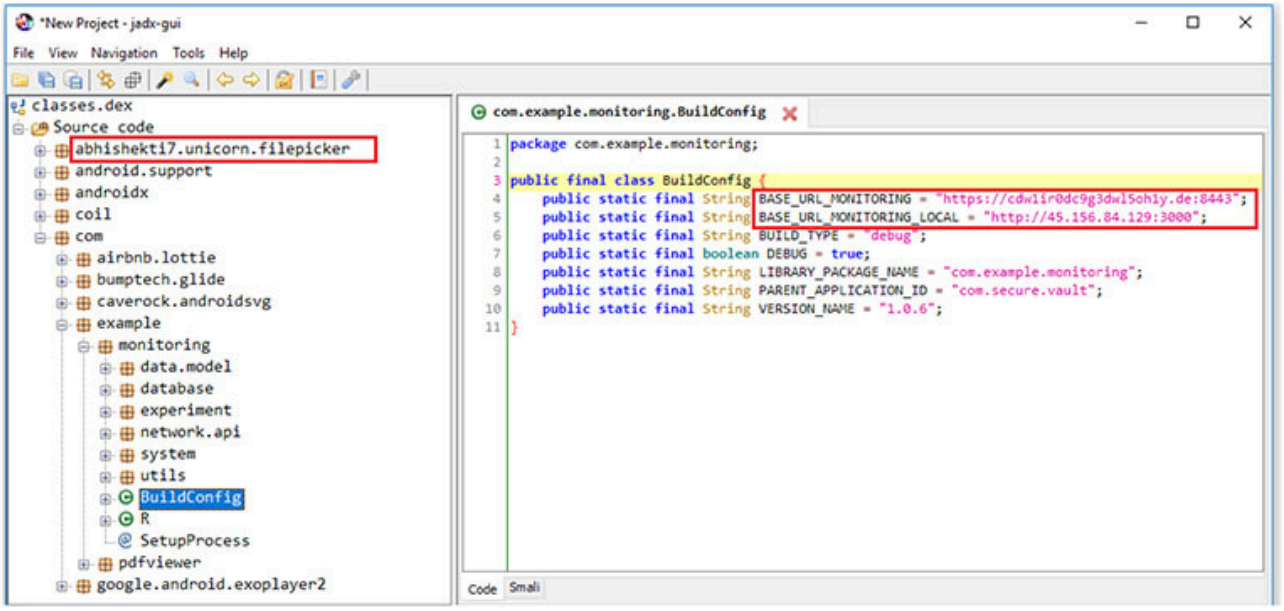


Figure 19. URL & IP.

List of IOCs

Sr.no	Indicator	Type	Remarks
1	45a6a0b2b02a9d288afba1ff41c689be9b9bd40ee862aa4bd6b036e3f0a4c3ab	SHA-256	SafeShare.apk
2	cdw1ir0dc9g3dwl5oh1y.de	DNS	C&C
3	http://45.156.84.129	IP	C&C

MITRE ATT&CKTM Techniques Detection

Sr.no	Tactics	Technique ID
1	TA0003: Persistence	T1398: Boot or Logon Initialization Scripts
2	TA0032: Discovery	T1418: Software Discovery T1420: File and Directory Discovery T1426: System Information Discovery
3	TA0009: Collections	T1430: Location Tracking T1429: Audio Capturing T1512: Video Capture T1417.001: Input Capture: Keylogging T1636.002: Protected User Data: Call logs T1636.004: Protected User Data: SMS Messages T1636.003: Protected User Data: Contacts List
4	TA0011: Command and Control	T1437: Application Layer Protocol: Web Protocols
5	TA0010: Exfiltration	T1646: Exfiltration Over C2 Channel

Conclusion

Bahamut is Iran’s state-sponsored Advance Persistent threat group, communicating in the Urdu language for strategic social engineering attacks. This indicates the threat actor knows its target very well and is focused to compromise the individual target by going a bit far. In another example, the threat actor injected code to track keystrokes entered on Conion. Conion is a Tor-based chatting app, that is currently considered to be an alternative to Signal, which is not well known to many, which again proves the threat actor is clear with their aim. The mobile campaign operated by the Bahamut APT group is still active. Further, the spyware code, and hence its functionality, is the same as in previous campaigns, including collecting data to be exfiltrated in a local database, before sending it to the operators’ server, a tactic rarely seen in mobile cyberespionage apps.
