Dissecting Android Malware - Post 2:

Mamont Variant - Advanced Dropper Logic and Encrypted Payload Reconstruction

Prepared by
Vamsi Pavuluri

Table of Contents

1. Introduction

In the first post[1], we analyzed a simpler Mamont banking trojan that installs a payload APK from data.bin in the res/raw directory. In this second post, we explore a more advanced variant that dynamically mutates based on runtime conditions and employs extensive decoys and anti-reversing mechanisms to hinder analysis and obscure its true execution flow.

1.1 Overview of the Sample

Primary APK Hash: 9df3cd9085714d229c0bb5df501794ae98c03adb8bcd0c813260b1f2b591c304
Distribution URL: https[:]//uzb-omaaad.top/OmadShou.apk
Package Name: com.cfqbtbck.vlksdcmo
Main Activity: jkecmo.com.Un5dmv0czz
Version: 12.0
Hashes of similar samples: 2329a6d93662926ebc05df7456fc2f2a9e84c048a68e3dd0df506e1e67a2c7ae and 6c642ee9cc1980d5d3a1be2a92b94c4ee0c90da2548a8bc23eed453e29cd7fe6.

2. The Dropper App

We began by analyzing the APK using JADX[2] for high-level decompilation and apktool[3] to disassemble the application into Smali. While JADX provided an initial overview, several critical sections failed to decompile, requiring deeper inspection of the corresponding Smali code.

The distribution URL hosts a website in Uzbek that strongly suggests social-engineering driven delivery, likely propagated through Telegram channels. Key phrases include:

- **“Omad Shou”** – Appears to be a branded application or campaign name
- **“rasmiy ilovasini yuklab oling”** – “Download the official application”
- **“Ilovani tez va xavfsiz yuklab olish uchun Telegram sahifamizga o‘ting”** – “To download the application quickly and securely, visit our Telegram page”
- **“Telegram’dan yuklab olish”** – “Download from Telegram”

[1] https://www.nccgroup.com/research/dissecting-android-malware-post-1-mamont-banking-trojan/
[2] https://github.com/skylot/jadx
[3] https://apktool.org/

Figure 1: *Website where malware was downloaded from*

The repeated references to Telegram indicate that the malicious APK is likely distributed directly through Telegram channels, a common tactic to bypass traditional app store security controls.

2.1 WebView-Based UI Injection

The assets directory reveals key files involved in UI delivery and payload staging.

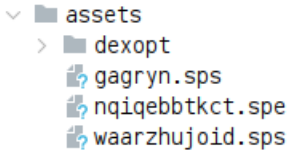

Figure 2: *Contents of assets folder*

The **gagryn.sps** and **nqiqebbtkct.spe** files function as **WebView resource containers**, holding encoded UI content used by the malware. These files are loaded at runtime to render phishing interfaces or overlays inside Android WebView components, allowing the malware to present fake login screens (e.g., banking pages) while keeping the actual payload logic separate and concealed.

```
public static void Gjiidqnxyecptzulvortrhfdjgjgsxgmxs(WebView webView, Un5dmv0czz un5dmv0czz, String str) {
    Lvykjdzgpnpmedglwtypvjkqvdudx.Hdtsjbvgjmggnrwhmfroy(str, "assetName");
    try {
        byte[] bytes = "562747852hxrIya7lnN2kU2SvrcfU5B6".getBytes(Facnfgjstfsyraerfvizkydoth.Gquesyvnltekgxdkymmis.Tnbvjiexzjuayqthripayhscknbauxf);
        Lvykjdzgpnpmedglwtypvjkqvdudx.Dbtlffbpbggmfwrlastdefhhglvkfshm(bytes, "getBytes(...)");
        BufferedInputStream bufferedInputStream = new BufferedInputStream(un5dmv0czz.getAssets().open(str), 65536);
        try {
            byte[] bArrTvemikwnqcoebfyyzill = Igmmzymaazkvykdianvycnkljes.Tvemikwnqcoebfyyzill(bufferedInputStream);
            Igmmzymaazkvykdianvycnkljes.Dbtlffbpbggmfwrlastdefhhglvkfshm(bufferedInputStream, null);
            byte[] bArr = new byte[bArrTvemikwnqcoebfyyzill.length];
            int length = bArrTvemikwnqcoebfyyzill.length;
            for (int i = 0; i < length; i++) {
                bArr[i] = (byte) (bArrTvemikwnqcoebfyyzill[i] ^ bytes[i % bytes.length]);
            }
            webView.loadDataWithBaseURL("file:///android_asset/", new String(bArr, Facnfgjstfsyraerfvizkydoth.Gquesyvnltekgxdkymmis.Tnbvjiexzjuayqthripayhscknbauxf), "text/html",
        } finally {
        }
    } catch (Exception unused) {
    }
}
```

Figure 3: *The code reads encrypted content from gagryn.sps and nqiqebbtkct.spe in the assets directory, applies XOR decryption using the hardcoded key 562747852hxrIya7lnN2kU2SvrcfU5B6, and loads the resulting HTML/JavaScript into a WebView.*

2.2 APK Reconstruction

The **Igmmzymaazkvykdianvycnkljes** class implements a full payload reconstruction routine that transforms an embedded, obfuscated archive into a usable APK. It begins by reading encrypted data from the asset (waarzhujoid.sps) and applying a repeating XOR decryption using a hardcoded key, after which the resulting data is treated as a ZIP archive and unpacked into a working directory.

This decoding process can be broken down into two stages. First, the contents of **waarzhujoid.sps** are decrypted using a repeating XOR operation with the static key (*562747852hxrIya7lnN2kU2SvrcfU5B6*). The decoded output is then decompressed using raw DEFLATE (Inflater without a zlib header), producing an intermediate APK (otter.apk). This intermediate file is subsequently processed by an additional routine that reconstructs or sanitizes the ZIP structure, resulting in the final payload APK (wolf.apk).

```
byte[] raw = readAsset("waarzhujoid.sps");

// Stage 1: XOR decode
byte[] decoded = decode(raw);

// Stage 2: raw deflate inflate
byte[] apkBytes = decompress(decoded);

// Write file
FileOutputStream fos = new FileOutputStream(outFile);
fos.write(apkBytes);
fos.close();
```

Figure 4: *Pseudo code illustrating the APK extraction flow*

We used the following Python code used to extract otter.apk file from waarzhujoid.sps

```
import zlib

key = b"562747852hxrIya7lnN2kU2SvrcfU5B6"

data = open("waarzhujoid.sps", "rb").read()

# Step 1: XOR
decoded = bytearray(len(data))
for i in range(len(data)):
    decoded[i] = data[i] ^ key[i % len(key)]

# Step 2: RAW DEFLATE (IMPORTANT: wbits=-15)
try:
    decompressed = zlib.decompress(bytes(decoded), -15)
    open("otter.apk", "wb").write(decompressed)
    print("otter.apk recovered")
except Exception as e:
    print("Decompression failed:", e)
```

Figure 5: *POC code to extract APK*

The unpacked contents are then modified, including changes to classes.dex, updates to the AndroidManifest (such as altering launcher aliases), and the addition of auxiliary files like key.json in the assets directory. Finally, the modified file structure is reassembled into a new APK by rebuilding the ZIP

format using standard ZIP headers and records, effectively producing a repackaged payload ready for execution.

2.3 APK Installation

The application installs the reconstructed payload APK using a coordinated interaction between its activity, service, and broadcast receiver components.

The main activity (**jkecmo.com.Un5dmv0czz**) acts as the entry point and controller for the installation flow. When launched, it retrieves the payload APK path. In its onResume method, the activity first checks whether the target payload package is already installed; if so, it immediately launches the installed application and terminates itself. If the payload is not present, the activity proceeds to initiate installation by resolving the APK file in the app's private storage, generating a FileProvider URI, and invoking the Android package installer UI. In this way, the activity serves as the orchestrator that either hands off execution to the already-installed payload or triggers the installation process when needed.

A dedicated receiver (**jkecmo.com.Receiver**) monitors installation broadcasts, handling status codes such as success, user confirmation, or failure, and implements retry logic and fallback flows as needed, eventually launching the installed payload and terminating the dropper. Meanwhile, a foreground service (**jkecmo.com.Ono2ko9n18**) maintains execution using a wake lock, ensuring the installation process persists across retries and background restrictions, effectively orchestrating a reliable dropper-to-payload hand-off.

2.4 Decoy Tactics

1. The malware intentionally sets the ZIP encryption flag on the reconstructed APK entries without actually encrypting the data, causing tools like unzip and apktool to fail or prompt for a password. This serves as an anti-analysis evasion technique to disrupt automated tooling and mislead analysts while leaving the payload fully accessible at runtime.

2. The logic*.sps files in Resources folder are decoy assets containing plain-text statistical or script-like content rather than executable or binary payload data. Their presence is a deliberate distraction technique, designed to mislead analysts into chasing irrelevant files while the actual malicious payload remains hidden in a separate encoded asset.

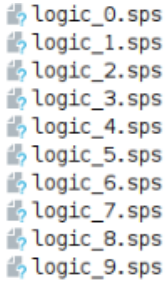

Figure 6: *sps files in Resources folder*

3. The extra AndroidManifest.xml and classes.dex observed in the non-standard (un-namespaced) folder appear to be decoy or intermediate artifacts rather than the actual payload. They likely serve to confuse analysis or represent partially reconstructed outputs during the loader's staging process but are not part of the final executable APK chain.

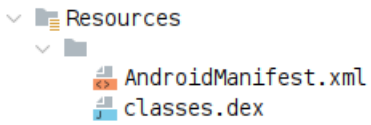

Figure 7: *Dex file and Manifest file in un-namespaced folder*

4. During dynamic analysis, the application was observed continuously updating entries in the f9k2m7n4q1p8.xml file under shared_prefs, with values rotating between different layout paths and package names across executions. These changes appear to serve as decoy or environment-sampling mechanisms, likely used to fingerprint the runtime environment and introduce variability to hinder repeatable analysis, rather than storing meaningful persistent configuration data.

```
Launch:1
<map><string name="ll7m4k9r3s1">res/layout/abc_search_dropdown_item_icons_2line.xml</string>
    <string name="lp3n8k5r2q9">eu.faircode.xlua</string></map>

Launch:2
<map> <string name="ll7m4k9r3s1">res/layout/abc_search_dropdown_item_icons_2line.xml</string>
    <string name="lp3n8k5r2q9">com.android.stk</string> </map>
```

Figure 8: *Varying values in shared_prefs after every launch*

2.5 Anti-Analysis Tactics

Crucially, the loader also includes a dedicated anti-analysis module that actively scans for dynamic instrumentation frameworks (e.g., Frida), hooking and rooting solutions (Magisk, Xposed, Substrate), Frida gadget binaries, and well-known instrumentation ports. If any such indicators are detected, execution silently aborts before the payload reconstruction stage, leaving the app permanently stalled in its decoy UI state.

3. Behavior of the Companion Malware

wolf.apk
Package Name: nkgkhka.com
Main Activity: nkgkhka.com.Zyoz7pbxzy
Version: 14

3.1 Dangerous Permissions Requested

- RECEIVE_SMS
- READ_SMS
- SEND_SMS
- READ_PHONE_STATE
- CALL_PHONE
- READ_PHONE_NUMBERS

These permissions allow total access to the user's SMS inbox, outgoing messages, phone identity, and the ability to make calls enabling fraud, account takeover, and covert communication via SMS.

3.2 Malware Capabilities and Command Handling

App has multiple high-risk capabilities, including **sending SMS messages**, **initiating phone calls and executing USSD requests**, **collecting installed applications**, **gathering device/network information**, and **manipulating system behavior to suppress notifications** (e.g., enabling Do Not Disturb). It also includes logic to **access SMS-related data**, potentially enabling interception or exfiltration of messages, and uses Android telephony APIs to interact directly with SIM cards and carriers. Additionally, it can **execute commands dynamically based on input**, indicating remote control functionality, and it reports execution results back through an internal handler or communication channel.

Command strings observed:

- sms
- apps
- call
- send
- check
- blockpush

4. Conclusion

This Mamont variant demonstrates how modern mobile banking malware has evolved through:

- Advanced multi-stage payload reconstruction (XOR + raw DEFLATE + repackaging)
- WebView-based phishing UI delivery separated from core payload logic
- Extensive anti-analysis and evasion techniques (decoys, fake ZIP encryption flags, environment fingerprinting)
- Resilient installation orchestration (activity, service, and receiver coordination with retry logic)
- Command-driven remote control capabilities enabling SMS, call, USSD, and data-collection operations

The dropper and its dynamically reconstructed companion APK operate together to decrypt, install, and execute the hidden payload while maintaining persistence and evading detection. By combining obfuscation, staged execution, and a flexible command-handling backend, this variant is capable of silently taking control of infected devices, harvesting sensitive data, and performing telecom-based financial fraud under attacker control.