**Wuta Camera App Security Analysis**

* **Objective** of this project was to perform a full security analysis of Wuta Camera Android App.

1. Static Analysis: MobSF, JADX
2. Dynamic Analysis: Burp Suite
3. Manual inspection

* **Why?**

- 10+ Million Downloads Worldwide

- Holds very sensitive data

- Previous Security reports

In 2024, Wuta integrated a third-party SDK to add features to their app, but that SDK was malicious and introduced security risks like ad fraud and silent JavaScript execution. It was found that this malware had already infected over 11 million devices worldwide.

SDK stands for **Software Development Kit**. It’s a bundle of tools, code, and libraries that developers use to add specific features to their apps — like ads, analytics, camera filters, etc.

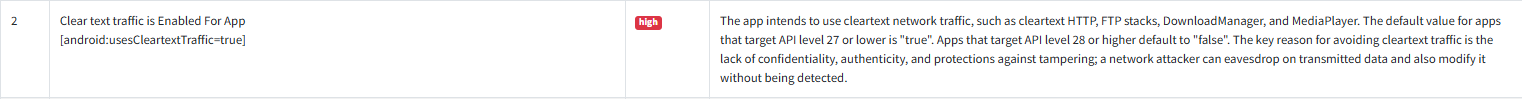
Turn out that the average mobile app uses around 30 SDKs, and 90% of code sourced from third parties. While this widespread use delivers significant benefits to developers, it also raises safety and security issues.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Vulnerabilities Report – Comparing MobSF & JADX**

1. **Cleartext Traffic Enabled**



MobSF's **static analysis** flagged the app for allowing **cleartext traffic** (HTTP instead of secure HTTPS). This is considered a **security weakness** because it allows data to be transmitted over an unencrypted channel, which can be intercepted.

**Confirmation –**

I decompiled the APK using JADX and manually inspected the **Manifest** file. I found this:

A screenshot of a computer code

AI-generated content may be incorrect.

* **Why is a Risk? -** Normally, apps are expected to use HTTPS to encrypt data in transit.
* Enabling usesCleartextTraffic=true allows communication with HTTP endpoints.
* If the app sends user data (e.g. logins, personal info) over these connections, it could be sniffed or intercepted by an attacker (especially on public Wi-Fi).

1. **Signatures**

MobSF static analysis report flagged that the APK is only signed with v1 and v2, not v3 or v4 signatures.

A close up of a certificate

AI-generated content may be incorrect.

without v3 signatures (added in Android 9.0), the app lacks support for key rotation, which is essential for securely updating the app over time. Without v4 signatures (added in Android 11), the app misses important performance and security optimizations on modern devices. This gap in security best practices could make the application more vulnerable to tampering attacks and suggests that other security measures in the app may also be outdated or insufficient by current standards.

A computer screen shot of a computer

AI-generated content may be incorrect.A screenshot of a computer

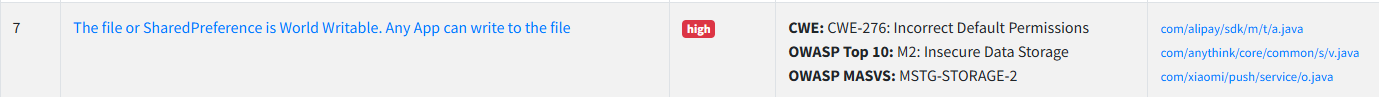
AI-generated content may be incorrect.I cross-verified it with JADX, using the apksigner:

**Why is a Weakness?**

* The app uses both v1 and v2 signatures. V1 is needed for Android 5 and 6 to install the app, but it makes the app vulnerable to known attacks like the Janus vulnerability. Janus is a flaw that lets attackers secretly change an APK without breaking its signature. On Android 7 and above, only v2 is checked, so disabling v1 would improve security without affecting newer devices. OWASP and Google practices are to disable v1 completely.

1. **Insecure Data Storage (M2)**

MobSF flagged possible storage of sensitive information in unprotected local storage (SharedPreferences), which is a common method Android apps use to store user data.



I manually confirmed with JADX locating the class *com.alipay.sdk.m.t.a*

Encountered this method:

A screen shot of a computer code

AI-generated content may be incorrect.

tid (a transaction/session ID)

client\_key

timestamp

vimei (virtual IMEI)

These values are wrapped into a JSON object and stored in the app’s **SharedPreferences**, like this:

**Why this is dangerous:**

SharedPreferences are stored in local files. If an attacker gains access (e.g. via root access, malware, or physical access), they can **read all this sensitive information**.

C0041a.d("alipay\_tid\_storage", "tidinfo", jsonObject.toString(), true);

The method d() then tries to encrypt the data using helper functions.

But the real problem is If encryption **fails** (which happens silently), the data is **still stored unencrypted**. No alerts, data saved in plaintext.

A screen shot of a computer code

AI-generated content may be incorrect.

1. **Improper Platform Usage (M1)**

**MobSF flagged this file because WebView debugging is enabled via WebView.setWebContentsDebuggingEnabled(true) in the AgentWebConfig class.**

****

I opened the class using JADX and confirmed the code is indeed there. If this debug() method is ever called on a real device, an attacker could connect via USB or Wi-Fi and access the WebView’s internals — leading to data leakage or manipulation. This is especially risky if the WebView handles login forms or payment data.

**A close up of text

AI-generated content may be incorrect.**

**Why is a Vulnerability?**

This setting enables Chrome DevTools to remotely inspect and control any WebView running in the app. While useful for development, it is a critical misconfiguration if left active in production.

**Why it’s dangerous?**

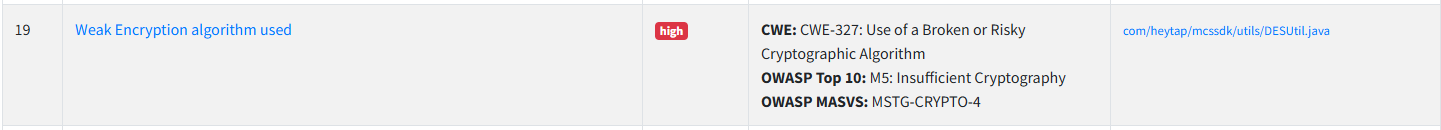
**Any attacker** with physical access or remote ADB access to a user's device can open Chrome and visit chrome://inspect, gaining full visibility of the app's WebViews.

This **completely bypasses the app’s UI controls** and can be used for phishing or abuse.

They can:

* View sensitive data (e.g. personal info, tokens, sessions)
* Inject or modify JavaScript
* Hijack sessions or manipulate form submissions

1. **Insufficient Cryptography (M5)**

****MobSF flagged usage of an outdated and insecure cryptographic algorithm — DES (Data Encryption Standard) — which is no longer considered secure by modern standards.

**I manually verified with JADX, located the class** *com.heytap.mcssdk.utils.DESUtil*

Found this:

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Vulnerability?**

DES is an outdated cryptographic algorithm. It was designed in the 1970s and officially deprecated in the early 2000s by the U.S. government (NIST) because it's too weak for modern use.

The main issue is that DES only uses a 56-bit key, and with today's computers, this can be brute-forced in hours or minutes. That means an attacker can try every possible key until they find the right one and decrypt your data.

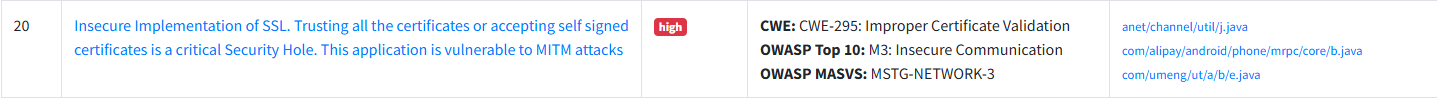
**Why is dangerous?**

**Any sensitive data encrypted with DES can be easily cracked**, including:

* Passwords
* Session tokens
* Personal user information

1. **Insecure Communication (M3)**

MobSF flagged the app for trusting all SSL certificates, including **self-signed ones**, without validation.

****

**After manually checking the file on JADX, I encountered this:**

**A screen shot of a computer

AI-generated content may be incorrect.**

The code sets up a secure connection (SSL/TLS) but has a big problem: it doesn't properly check who it's talking to.

Think of it like this - you're making a phone call to your bank, but you don't verify it's the bank that answered. You just trust whoever picks up.

**Why is dangerous?**

* **Anyone Could Pretend to Be the Server**: If someone gets between you and the real server (like on public WiFi), they could pretend to be the real server.
* **Your App Would Be Fooled**: Without proper checks, your app would connect to the fake server thinking it's talking to the real one.
* **Your Data Could Be Stolen**: You might send private information (passwords, personal data) to the attacker thinking you're sending it to the trusted server.

**Dynamic Analysis – Burp Suite & MobSF**

**A white line on a white surface

AI-generated content may be incorrect.**

At runtime, the app continues to send sensitive API calls over HTTP and shows incomplete TLS pinning—leaving it open to man-in-the-middle attacks and eavesdropping.



A screenshot of a computer

AI-generated content may be incorrect.

**ID swap (151 → 150)**

* **Goal:** My goal was to see how this code handles bad id’s
*  **Result:** I still got back the exact same tree.
*  **Takeaway:** Rather than tell me “room 150 doesn’t exist,” the server quietly shows me “room 151.”
* **What it means:** The service doesn’t actually look up “ID 150”—it silently falls back to the default (151). That’s not a big vulnerability, but it tells us the code is doing something like “if you ask for a missing tree, just serve the default one.”

**Security flag?**

* **Low impact.** You didn’t get extra data, but you learned the app falls back to a default rather than erroring.
* It’s a clue to how the code handles bad IDs, but not a direct leak by itself.

**Fetching the full JSON content**

* **Goal: Instead of asking “what’s the MD5 of that tree file?”, I asked “give me the actual JSON content of that tree.”**
* **Result: I got a full dump of the app’s internal configuration (menus, stickers, mosaic settings, etc.).**
* **Takeaway: The endpoint isn’t just for checksums — it’ll serve *any* JSON file it can read.**

**Security flag?**

* **High impact. You’re now seeing private app data that isn’t meant for public use.**
* **If you combine this with a path-traversal trick (pointing at ../../../etc/passwd.json), you could exfiltrate *any* file on the server.**

A screenshot of a computer

AI-generated content may be incorrect.

**What to conclude/report**

* **Test case**: tried ' OR '1'='1 in the um\_device\_token field of POST /api/notice/report\_device\_token.
* **Result**: 200 OK, normal “OK” response, no error or deviation.
* **Interpretation**: input is being safely handled (likely parameterized/escaped), so no SQLi here.

In conclusion, I felt like I was able to learn security analysis procedures that allowed me to get the previous results. It was a great experience that made me gain new skills and gave confident for future projects.