

**Name:** Hind Abu Alamrain  
**Student ID:** 300403234  
**Group Number:** F25\_4440\_G1

## 1. Description of Work Done

### So far, I did:

I have successfully conducted a **runtime forensic analysis** of the **Instagram Android application** using **Frida** and **Burp Suite**. My work focused on setting up a **rooted Android emulator**, bypassing **SSL pinning**, and capturing **HTTPS network traffic** between the app and its backend APIs.

The process involved preparing the test environment by launching a **rooted Android Virtual Device (AVD)**, installing **Python** and **Frida tools**, and setting up the **Instagram APK**. I deployed the **Frida server** on the emulator, configured **Burp Suite** as a proxy, and installed its certificate into the **system trust store** using **Magisk** and **RootAVD**. After that, I ran the **Frida SSL-bypass script** to intercept secure traffic and monitored the captured **HTTP/HTTPS communications** through **Burp Suite**.

Through this process, I extracted runtime data such as **device identifiers**, **session tokens**, **navigation chains**, **API endpoints**, and **account-related metadata** (including **usernames** and, in some cases, **email or phone number fields** associated with the logged-in account). These artifacts provided a deeper understanding of how the Instagram app communicates internally during user login and authentication flows, and helped document forensic evidence of user-to-server interactions.

Finally, the intercepted data was formatted and converted into structured **JSON records**, highlighting key forensic fields such as **device identifiers**, **session tokens**, **account information**, **timestamps**, and **network context** for clear documentation and reporting.

### **The issues I encountered:**

During development, I faced several technical challenges, including:

- Compatibility issues between **Frida** and the target application, leading to frequent app crashes.
- **SSL pinning** mechanisms that initially prevented traffic interception.
- **TLS handshake failures** caused by incorrect certificate authority (CA) installations.
- **Instagram** and **Telegram** occasionally failed to open or log in properly during testing, or crashed unexpectedly.
- The **Telegram** app repeatedly failed to log in, likely due to proxy or certificate-trust configuration problems.

I resolved these issues for **Instagram** by testing various **bypass scripts**, reinstalling system CAs using **RootAVD + Magisk**, and thoroughly validating all **proxy and certificate settings**. Work on the **Telegram** testing phase is still in progress.

### **These are my planned next steps:**

- **Repeat** the runtime forensic analysis process for the **Telegram** app, capturing and documenting all intercepted requests and responses.
- **If** Telegram continues to fail during testing, **switch** to **Twitter** or **Snapchat** as the second target application.
- **Continue researching Frida hook Java methods** for potential implementation in the second app, **if time permits**.

## **2. AI Use Section:**

<b>AI Tool Name</b>	<b>Version, Account Type</b>	<b>Specific feature for which the AI tool was used</b>
ChatGPT (GPT-5 Thinking mini)	Plus (paid)	<ul style="list-style-type: none"><li>- Troubleshooting app crashes.</li><li>- Understanding tool functions and new concepts.</li></ul>

## **3. Value Addition:**

While AI support provided background information and conceptual guidance, it was less effective for hands-on troubleshooting of app crashes. Therefore, additional research was conducted using technical articles, online tutorials, and multiple experimental trials with various emulators, APK versions, and certificate configurations until successful interception was achieved.

## **4. Appendix:**

what is a proxy.

**ChatGPT said:**

Short answer: a **proxy** is a middleman between your device and the internet. Your device sends requests to the proxy, the proxy forwards them to the server, gets the response, and sends it back to the device. This lets you *observe, modify, or record* the traffic.

**Why use a proxy for mobile testing**

To see full HTTP/HTTPS requests and responses (headers, bodies, cookies).

To modify requests/responses on the fly (for testing).

To use tools with a GUI (Burp Suite, mitmproxy) that make analysis easier than plain text logs.

To capture traffic from apps you can't instrument with Frida.

**Typical tools (popular)**

**Burp Suite** — GUI, feature-rich, common for security testing.

**mitmproxy** — command-line + web UI, scriptable in Python.

**Charles, Fiddler** — other options.

**HTTPS / SSL: the CA problem**

For HTTPS, the proxy acts as a TLS man-in-the-middle. To decrypt TLS you must trust the proxy's CA on the device (install the proxy's CA certificate) **or** bypass the app's SSL-pinning with Frida.

If the app uses SSL pinning and you don't bypass it, it will reject the proxy TLS cert and fail.

If you use Frida to bypass pinning, you can route traffic through a proxy **without** installing the CA (but installing CA + using proxy is still common).

**When you don't need a proxy**

If you can hook the app with **Frida** and the scripts extract all network details you need (headers, URLs, tokens). Frida captures at runtime inside the app—no proxy needed.

**How to set a proxy for an Android emulator (quick)**

**Simple emulator startup** (if using emulator binary):

```
emulator -avd Android_35 -http-proxy http://127.0.0.1:8080
```

**Or set proxy via adb for Wi-Fi:**

```
adb shell settings put global http_proxy 127.0.0.1:8080
```

To remove:

```
adb shell settings put global http_proxy :0
```

(If using an emulator image with Google Play, you may need to install CA cert to System or use Frida to bypass pinning.)

### **Summary / Recommendation for your project**

If you simply want network artifacts (tokens/headers/URLs) and you already have Frida + SSL bypass script → **no proxy required**. Spawn the app with Frida and capture.

If you prefer a GUI (Burp) to inspect raw HTTP/HTTPS flows, use a proxy + install the proxy CA on the emulator (or use Frida to bypass pinning).

---

### **What is a Magisk module?**

A **Magisk module** is a package (ZIP) that modifies an Android system *without* permanently changing the system partition — it applies changes “systemlessly” through Magisk. Modules can add files, change configuration, install hooks, or provide functionality (for example root utilities, UI tweaks, or security-related tools).

### **Key points — what it does**

Installs **system-level changes** (files, libraries, scripts) while keeping the real /system untouched.

Loads changes at boot via Magisk’s manager/service, so modules take effect automatically after reboot.

Can enable features that would otherwise require remounting or editing /system (e.g., add a system CA, install drivers, inject libraries).

Many modules support **Zygisk** which lets modules run code inside Android’s zygote (useful for runtime hooks).

### **Typical contents of a module**

system/ or system\_root/ tree with files to be merged into the system view.

META-INF/com/google/android/update-binary (installer script).

Optional scripts to run on install or boot.

### **Example: *AlwaysTrustUserCerts* module**

Purpose: copy user-installed CA certificates into the system CA store so apps treat them as trusted system CAs.

Useful for MITM inspection (mitmproxy/Burp) where apps would otherwise reject user CAs.

Requires Magisk because it modifies the system CA store in a systemless way.

### **Pros & cons**

Pros:

Systemless (safer to revert).

Easy to enable/disable via Magisk Manager.

Modules can survive OTA updates better than manual edits (depending on update).

Cons:

Requires Magisk (root + Magisk installed).

Some modules (or Magisk itself) may interfere with SafetyNet or other protections.

Not all devices/AVDs support the easiest install methods (older/newer Android changes).

### **Security note**

Installing modules gives code elevated privileges at boot — only install trusted modules from reputable sources and audit what they change.

---

Want to continue?

If you want to proceed now I can either:

Give the single command to copy your proxy CA into the emulator's **system** store (manual method you chose), **or**

Show how to install the AlwaysTrustUserCerts module on a Magisk-enabled device (if you later use a Magisk AVD).

Which do you prefer? Reply **System CA** or **Magisk module**.

**You said:**

Show how to install the AlwaysTrustUserCerts module on a Magisk-enabled device (if you later use a Magisk AVD).

**ChatGPT said:**

Nice — here's a short, exact how-to for installing **AlwaysTrustUserCerts** on a **Magisk-enabled device (or Magisk AVD)**. I'll give both the **easy UI method** and a **CLI/adb method** (if you prefer to push/install without touching the UI). Finish each small step before moving to the next.

**Install AlwaysTrustUserCerts (UI method — easiest)** Download the module ZIP (example project: *NViSOsecurity / MagiskTrustUserCerts* or a similarly-named ZIP). Put it on your PC.

Push the ZIP to the device (one command):

```
adb push "C:\path\to\AlwaysTrustUserCerts_v1.3.zip" /sdcard/
```

Open Magisk / Modules on the device.

Tap Install from storage → browse to /sdcard/AlwaysTrustUserCerts\_v1.3.zip → select it.

Allow Magisk to install the module. When install completes, Reboot the device.

After reboot, open Magisk → Modules and confirm Always Trust User Certificates is listed and enabled (toggle on).

Install AlwaysTrustUserCerts (adb / shell method)

Use this if you cannot/choose not to use the Magisk UI.

Push the ZIP to the device:

```
swift
```

Copy code

```
adb push "C:\path\to\AlwaysTrustUserCerts_v1.3.zip" /sdcard/
```

From a shell, become root and install the module using Magisk's installer script (if Magisk supports the CLI):

```
bash
```

Copy code

```
adb shell  
su  
  
# install via Magisk Manager installer (kitsune Magisk uses different helpers).  
  
# Preferred: use Magisk Manager UI. If you must use shell, unzip into modules dir:  
  
mkdir -p /data/adb/modules/AlwaysTrustUserCerts  
  
unzip -o /sdcard/AlwaysTrustUserCerts_v1.3.zip -d  
/data/adb/modules/AlwaysTrustUserCerts  
  
# create module.prop if not present (module zip usually contains it).  
  
chown -R root:root /data/adb/modules/AlwaysTrustUserCerts  
  
chmod -R 755 /data/adb/modules/AlwaysTrustUserCerts  
  
exit  
  
exit
```

Reboot the device:

nginx

Copy code

adb reboot

Note: The module zip usually contains the right structure. Unzipping into /data/adb/modules/<name> is the manual equivalent to “install from storage.”

What the module does (short)

On boot it copies user-installed CA certificates into the system CA store so apps that only trust system CAs will also trust your proxy CA (mitmproxy/Burp).

Works systemlessly via Magisk (no permanent system partition edits).

How to verify it worked (3 quick checks)

Run these after reboot:

Confirm module is active in Magisk UI (visual check).

List module folder:

```
adb shell su -c "ls -l /data/adb/modules | grep AlwaysTrustUserCerts"
```

Confirm the CA was copied into system CAs (module usually copies user certs into /system/etc/security/cacerts/ or the system view):

nginx

Copy code

```
adb shell su -c "ls -l /system/etc/security/cacerts | tail -n 20"
```

Look for your CA (filename is the hash + .0 or .pem depending on distro).

Optional: check that the certificate subject matches:

nginx

Copy code

```
adb shell su -c "grep -a -A2 'Subject:' /system/etc/security/cacerts/* | head -n 20"
```

Or pull the cert and inspect:

php-template

Copy code

```
adb pull /system/etc/security/cacerts/<filename> .
```

```
openssl x509 -in <filename> -noout -subject -issuer -fingerprint
```

Troubleshooting (common issues & fixes)

Module not listed after reboot — re-install via Magisk UI; ensure Magisk is installed/active (Zygisk on for modules requiring it).

App still rejects proxy certs — some apps pin certificates at app-level. Use Frida pin-bypass in addition to the module (recommended).

Module install fails on newer images — if Magisk can't patch boot (Android 34+), use a Magisk-enabled AVD or manually copy CA to /system/etc/security/cacerts (you have root, so manual copy is an alternate).

Permissions wrong — ensure cert files in /system/etc/security/cacerts have owner root:root and permissions 644 (chmod 644).

If you want, I can give the exact adb commands for your current setup (you already pushed AlwaysTrustUserCerts to /sdcard/ earlier). Tell me one choice:

A — Walk through the Magisk UI install (I'll give next single UI step).

B — Run the manual unzip → move → reboot (I'll give the exact adb commands now).

Reply A or B.

You said:

A

ChatGPT said:

One tiny step — do this in the emulator now and then tell me what you see (or paste the success message):

Open Magisk Delta on the emulator.

Tap Modules.

Tap Install from storage.

Browse to /sdcard/AlwaysTrustUserCerts\_v1.3.zip and select it.

Tap Install (or LET'S GO).

Wait for the installer to finish and tell me the exact final message shown (e.g., “Install successful” or any error).

---

Nice clear description — thanks. You've already done most of the hard work and shown the right evidence (logs, Burp, Frida output). Below I'll explain **why login after injection often breaks**, what's happening under the hood, and give **exact, practical steps** to try so you can reproduce a reliable demo for your report.

---

**When we inject the Frida JavaScript after logging into Instagram, the injection and interception work. But if I log out and then try to log in again after injection, the app pages become unavailable. The issue only occurs when attempting to log in after injection; if Instagram is already open and logged in, then starting the injection works.**

**If you inject (Frida) before doing a login, Instagram often performs extra checks (native pinning, device attestation, native TLS libs, QUIC/HTTP3, or integrity checks) on the new session and the app fails the checks → pages unavailable.**

**If you attach after the app is already logged in**, Frida only needs to bypass Java SSLContext for subsequent calls and it works more reliably.

For a reliable demo: **do two demonstrations** in your report:

**Attach-after-login** (best for showing captured tokens/requests reliably).

**Spawn-under-Frida-before-login** (show that it may succeed or fail and explain why — that demonstrates the forensic/anti-tamper behaviours).

### Why logging in after injection tends to fail (the technical reasons)

#### Early native checks & attestation

Instagram (and FB apps) run native security checks and attestation (Play Integrity / proprietary attestation) very early in the login flow. If Frida is present or if you hook things unexpectedly, the attestation or nonce retrieval can fail and the server rejects/blocks the login — app shows “Page isn’t available.”

#### Native TLS / non-Java stacks

Some network code uses native libraries (Conscrypt, BoringSSL, bespoke C++ code) which don’t go through the Java SSLContext path the Frida JS script hooks. The Java-level bypass will not affect those native checks.

#### Timing / race conditions

If you spawn the app under Frida but the app initializes network components before the Frida hooks are active, the pinning remains enforced for the first requests (login). Conversely, if you attach after those components init, you can bypass only later traffic.

#### QUIC / HTTP/3 / UDP fallback

Login flows may prefer QUIC/HTTP3 (UDP) which may bypass Burp/proxy or behave differently; when combined with failed attestation the page fails.

#### Intercept blocking

If Burp’s Intercept is ON and you don’t forward, the app will hang. That explains “page waiting” behavior during manual interception.

Practical recommended workflow (reliable, repeatable for your demo & report)

#### A — *Reliable & simple demo (recommended for presentation)*

Launch Instagram normally and log in **without Frida**.

Confirm app is functional in emulator.

Start frida-server on the emulator and start Burp (listener + CA pushed).

attach with Frida to the running Instagram process (not spawn). Example:

```
# list processes
```

```
"%USERPROFILE%\AppData\Roaming\Python\Python312\Scripts\frida-ps.exe" -Uai
```

```
# attach (example)
"%USERPROFILE%\AppData\Roaming\Python\Python312\Scripts\frida.exe" --
codeshare pcipolloni/universal-android-ssl-pinning-bypass-with-frida -U -p <PID>
```

Turn **Burp Intercept OFF**, browse in-app; Burp will now capture API calls and show JSON/tokens.

Result: Reliable capture of requests and tokens for demonstration.

#### B — *Spawn-under-Frida-before-login (show limitations and reasoning)*

This is the “try to bypass pinning for the login session” path. Use it to show what *can* and *can’t* be intercepted and to discuss why.

Make sure frida-server is running on the emulator.

Ensure Burp CA is pushed to /data/local/tmp/cert-der.crt.

Spawn Instagram under Frida **and do not pause** so app continues:

```
"%USERPROFILE%\AppData\Roaming\Python\Python312\Scripts\frida.exe" --
codeshare pcipolloni/universal-android-ssl-pinning-bypass-with-frida -U -f
com.instagram.android --no-pause
```

Watch Frida output; wait for:

```
[o] App invoked javax.net.ssl.SSLContext.init
[+] SSLContext initialized with our custom TrustManager!
```

Then try to do the login in-app. If it hangs or the page becomes unavailable, capture adb logcat & show it in your report — it demonstrates the attestation/native-check failure.

**If spawn fails to allow login**, explain that in the report as an expected limitation: Instagram validates device/process state on login and can reject connections if it detects instrumentation or fails attestation.

#### **Concrete troubleshooting commands to capture evidence (logs to include in report)**

Run these while the failed login happens — include outputs in your project report as evidence.

```
# 1. Confirm proxy
adb -s emulator-5554 shell settings get global http_proxy

# 2. Confirm Frida server and processes
adb -s emulator-5554 shell "ps -A | grep frida"
"%USERPROFILE%\AppData\Roaming\Python\Python312\Scripts\frida-ps.exe" -
Uai
```

```
# 3. Capture adb logcat while reproducing the login failure (filter)
adb -s emulator-5554 logcat -v time > instalogcat.txt

# 4. Tail Frida server log if present
adb -s emulator-5554 shell "tail -n 200 /data/local/tmp/frida-server.log || echo no-frida-log"

# 5. If QUIC suspected, show iptables/udp blocks
adb -s emulator-5554 shell "iptables -L -n -v"
```

Include relevant logcat fragments in your report — they are the best evidence of attestation or native failures.

---

Things to try if you must demo a login-under-injection (advanced)

*(note: more intrusive/risky and not guaranteed)*

**Use a rooted emulator/image that allows adb root** and ensure frida-server runs as root so it can hook native libs earlier.

**Block UDP/443** on emulator to force HTTP/2 over TCP (so Burp can intercept).  
Example:

```
# on emulator shell (blocks UDP port 443)
adb -s emulator-5554 shell "iptables -A OUTPUT -p udp --dport 443 -j DROP"
```

This may force TCP fallback (less likely to use QUIC), but servers sometimes refuse fallback during login.

**Hook native functions** (libssl/BoringSSL) with Frida scripts (harder: you need platform-specific native hooks).

**Repackage app with Frida gadget** to ensure hooks active earlier (invasive and may violate terms; treat carefully).

---

### Final quick checklist for your immediate next run (copy-paste)

Start Burp listener, export CA (DER) -> PortSwiggerCA.cer.

Push CA:

```
adb -s emulator-5554 push PortSwiggerCA.cer /data/local/tmp/cert-der.crt
```

Start frida-server on emulator (as root):

```
adb -s emulator-5554 root
adb -s emulator-5554 remount
```

```
adb -s emulator-5554 push frida-server /data/local/tmp/frida-server  
adb -s emulator-5554 shell "chmod 755 /data/local/tmp/frida-server"  
adb -s emulator-5554 shell "/data/local/tmp/frida-server &"
```

Set global proxy:

```
adb -s emulator-5554 shell settings put global http_proxy 10.0.2.2:8080
```

Login-first demo: login normally; then attach and capture:

```
frida-ps.exe -Uai  
frida.exe --codeshare pcipolloni/universal-android-ssl-pinning-bypass-with-frida -U -  
p <INSTAGRAM_PID>  
# Intercept OFF in Burp, then use app -> capture requests
```

Spawn-under-frida demo (if you want to show attempt before login):frida.exe --  
codeshare pcipolloni/universal-android-ssl-pinning-bypass-with-frida -U -f  
com.instagram.android --no-pause

```
# wait for SSLContext init message, then attempt login and capture logcat  
adb -s emulator-5554 logcat -d | findstr /C:"AndroidAppAttestationPluginKeystore"  
/C:"SSLContext" /C:"FATAL EXCEPTION"
```

---

If you want, I'll now:

produce a **one-block script** (Windows batch) with the exact commands you used (for your README), or

draft the short “**Limitations**” paragraph (1–2 paragraphs) you can paste into your report explaining why login-under-injection fails and why you did the attach-after-login demo.

---

## What is QUIC / HTTP/3

**QUIC** is a modern transport protocol that runs over **UDP** instead of TCP. It was designed by Google and then standardized by the IETF. QUIC moves many things that used to live in TCP (connection management, multiplexing, loss recovery) into the transport layer and combines that with **TLS 1.3** for encryption right inside the transport.

**HTTP/3** is simply HTTP running on top of QUIC (analogous to HTTP/2 on TCP). So when a client says “HTTP/3”, under the hood it’s speaking QUIC + HTTP semantics.

Key QUIC/HTTP/3 properties you should know:

**Runs on UDP**, not TCP.

**Built-in TLS 1.3** — TLS handshake is part of QUIC, not a separate layer over TCP.

**Encrypted headers / metadata** (many parts that were visible on TCP are now protected).

**Multiplexing without head-of-line blocking** (better perf than HTTP/2 on TCP).

**Connection migration** (IP/port changes don't break the QUIC connection).

**RTT resumptions** possible (faster re-connections but replay/caching implications).

**SNI / hostname exposure can be reduced** by TLS extensions (ECH), so visibility of destination host may be limited.

Why this matters for Burp / Frida interception

### For a proxy-based interceptor (Burp / mitmproxy style)

**Different transport (UDP):** Traditional HTTP interceptors expect TCP (and a TLS stream they can terminate). QUIC is UDP-based and integrates TLS. A TCP/TLS MITM (generate a fake cert and have client trust Burp's CA) **does not** automatically intercept QUIC traffic.

**No separate TLS stream to terminate:** Because TLS is inside QUIC, you can't simply open a TCP listener and MITM the TLS handshake the old way. The MITM proxy must implement QUIC (including its TLS1.3 integration) to speak QUIC with the client and then re-establish another QUIC/TCP/TLS session with the server — i.e., it must be QUIC-aware and perform full protocol translation/termination.

**Encrypted metadata / ECH:** Even if you capture packets, less information (hostnames, headers) will be visible on the wire. That reduces what passive sniffing can show.

**Connection migration & 0-RTT:** Makes some replay and session-tracking approaches trickier for proxies.

**Practical consequences:** If your tool (e.g., Burp Community/Pro) doesn't implement QUIC termination, clients will keep using QUIC and you'll see no HTTP traffic to intercept. Workarounds include forcing the client to use HTTP/1.1 or HTTP/2 (disable QUIC in the client or application), using a QUIC-capable proxy, or instrumenting the client app to downgrade.

### For instrumentation-based interception (Frida, hooking)

**Frida hooks the app runtime**, not the network wire — so QUIC doesn't block you *per se*. If you can hook the HTTP/QUIC library functions (e.g., high-level HTTP request functions or lower-level socket/send/recv or native QUIC library APIs), you can inspect/modify requests before they hit QUIC.

**Library differences matter:** Many apps use high-level HTTP client libs (OkHttp, URLSession, curl, libquic, msquic). You may need different Frida hooks depending on the library and whether the QUIC implementation is native or in managed code.

**Certificate pinning and in-process TLS:** QUIC uses TLS 1.3 inside the process; if the app pins certificates or validates TLS in-process, you'll need to bypass or hook that verification — same as for TLS over TCP, but the verification points may be different because they're in QUIC/TLS integration code.

**Advantage:** Frida can make the client talk plain HTTP (or give you the plaintext) before it's encrypted by QUIC — often the most reliable way to see real request/response contents without needing a QUIC-aware proxy.

Practical options / recommendations (concise)

**If you control the client (emulator/device/app):**

Easiest: **disable QUIC** or force HTTP/2/1.1 in the client so Burp can intercept as usual.

If you can't disable QUIC: use **Frida to hook the HTTP client** (dump requests/responses or force a downgrade).

**If you must proxy QUIC traffic:**

Use a **QUIC-aware proxy/terminator** (requires the proxy to implement QUIC + TLS termination). Not all proxies do this.

Alternatively, set up a transparent solution that translates QUIC→HTTP/2 or capture/decode QUIC at the packet level (hard, needs protocol-savvy tooling).

**If app uses certificate pinning:** Frida hooking to bypass pinning or hooking verification functions is often easier than trying to make a proxy appear trusted to the app.

**If you only have packet captures:** Expect to see UDP QUIC packets with encrypted payloads — you'll get limited readable data unless you have session keys from the client.

TL;DR

QUIC/HTTP3 moves transport + TLS into a UDP-native, encrypted protocol. That **breaks traditional TCP/TLS MITM proxies** (like classic Burp setups) because there's no separate TLS stream to terminate. **Frida** (runtime instrumentation) remains very useful because it can inspect or change the app's data **before** QUIC/TLS encrypts it, but you'll need to hook the right library or bypass pinning.

TLS?

**ChatGPT said:**

Short answer: **TLS = Transport Layer Security** — it's the encryption layer that protects web/mobile traffic (what makes HTTPS secure). Below is a compact, project-

focused explanation so you know exactly why TLS matters for your Frida + Burp workflow.

### **What TLS does :**

TLS encrypts traffic between the app and the server so eavesdroppers cannot read or modify requests/responses.

### **Key TLS concepts you need**

**Handshake:** client and server agree on encryption keys and certificates before sending data.

**Certificate:** proves the server's identity (issued by a CA).

**CA (Certificate Authority):** trusted issuers whose root certs are in the system/user store.

**Certificate pinning:** the app checks the server certificate or its public key and refuses other certs even if signed by a trusted CA.

### **Why TLS matters for your project**

Instagram's traffic is TLS-protected, so a plain proxy cannot read bodies unless the TLS connection is *broken* and re-established via a trusted proxy certificate (MITM).

If the app uses **certificate pinning**, it will detect the proxy's CA and refuse the connection, so Burp alone won't see decrypted content.

Two ways to see decrypted HTTPS in your lab

### **MITM Proxy (Burp/Charles) + trusted CA**

Install the proxy CA into the device **system** trust store (requires root/Magisk for Android).

App then accepts the proxy cert and Burp can decrypt TLS.

Advantage: reliable view of full requests/responses.

Disadvantage: requires root/system changes or the app must not pin certificates.

### **Frida SSL-pinning bypass (no system CA required)**

Hook and override the app's pinning checks (Java-level or native).

After bypass, the app accepts the proxy cert or you can log plaintext at the point where request is built.

Advantage: can avoid modifying system CA.

Disadvantage: fragile (app detects Frida, or pinning in native code can be harder to bypass).

### **Why Frida-only sometimes fails vs why proxy helps**

**Frida-only works** if you can hook the function where the app still has plaintext (e.g., OkHttp Request.Builder.build()), or if you successfully bypass pinning at Java/native layer.

**Frida-only fails** when app does TLS/pinning in native libraries or you can't find the right hook. Then a proxy + system CA (or Frida bypass) is the more reliable approach.

### **Quick troubleshooting checklist (TLS-related)**

Match frida-server version to frida-tools.

If Burp shows TLS errors: either install proxy CA to system store **or** run a Frida bypass.

If app crashes after Frida: try different bypass scripts or fewer/targeted hooks.

If nothing hits Burp: confirm emulator proxy settings and that the app respects system proxy (some native stacks bypass it).

---

## **SSLContext.init — beginner-friendly explanation**

### **What is SSLContext?**

SSLContext is a Java class that represents the configuration for TLS/SSL connections. It contains the cryptographic setup used when a client (your app) opens a secure connection to a server: which keys, which trust managers (which CAs to trust), and random sources.

### **What does SSLContext.init(...) do?**

SSLContext.init(km, tm, sr) sets the actual components used by that TLS context:

km = KeyManagers (identities/certificates you present to servers) — often null for clients.

tm = TrustManagers (which certificate authorities the client trusts).

sr = SecureRandom (randomness source).

After init runs, any sockets created from that SSLContext will use the specified TrustManagers and KeyManagers.

### **Why is it important for pinning/mitm?**

If an app uses custom TrustManager or pins certificates, it will usually create an SSLContext with a TrustManager that only accepts certain certificates.

By replacing what init installs as TrustManagers, you can make the app trust your CA (or a permissive TrustManager) — this is the basis of Java-level SSL-pinning bypass.

### Small Java example (how apps might use it)

```
// app code: configure SSLContext to trust only certain CAs
TrustManagerFactory tmf =
    TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType()); // ... load keystore with
pinned cert ...
tmf.init(ks);
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(null, tmf.getTrustManagers(), new SecureRandom());
SSLSocketFactory
factory = ctx.getSocketFactory(); // use factory to make HTTPS connections
```

If you can change the tmf.getTrustManagers() argument, you change what the app trusts.

### How Frida uses this to bypass pinning (concept)

**Frida can hook SSLContext.init(...) and replace the tm argument with a TrustManager array built from your own KeyStore (containing Burp CA), or with a permissive TrustManager that accepts any cert.**

Since this happens *at runtime*, the app will use your TrustManagers instead of the app's pinned ones — enabling MITM.

### Beginner-friendly Frida hook example (JavaScript)

This is a compact example that **overrides** SSLContext.init(...) to inject a permissive TrustManager. Run the script early when launching the app.

```
Java.perform(function () {
    try {
        var SSLContext = Java.use('javax.net.ssl.SSLContext');

        // Overload with standard signature: (KeyManager[], TrustManager[],
        // SecureRandom)
        SSLContext.init.overload(['Ljavax.net.ssl.KeyManager;',
            'Ljavax.net.ssl.TrustManager;', 'java.security.SecureRandom'])
            .implementation = function (keyManagers, trustManagers, secureRandom) {
                // create a permissive TrustManager that accepts all certs
                var X509TrustManager = Java.use('javax.net.ssl.X509TrustManager');
                var PermissiveTrustManager = Java.registerClass({
                    name: 'org.frida.PermissiveTrustManager',
                    implements: [X509TrustManager],
                    methods: {
```

```

checkClientTrusted: [ { args:
  ['[Ljava.security.cert.X509Certificate;','java.lang.String'], retval: 'void',
  implementation: function(a,b) {} }],
  checkServerTrusted: [ { args:
  ['[Ljava.security.cert.X509Certificate;','java.lang.String'], retval: 'void',
  implementation: function(a,b) {} }],
  getAcceptedIssuers: [ { args: [], retval: '[Ljava.security.cert.X509Certificate;',
  implementation: function() { return Java.array('java.security.cert.X509Certificate',
  []); } } ] } ]);

var permissiveArray = Java.array('javax.net.ssl.TrustManager',
[PermissiveTrustManager.$new()]);
// call original init but inject our permissive TrustManager
return SSLContext.init.overload('[Ljavax.net.ssl.KeyManager;',
'[Ljavax.net.ssl.TrustManager;','java.security.SecureRandom')
.call(this, keyManagers, permissiveArray, secureRandom);
};

send('[frida] Hooked SSLContext.init and injecting permissive TrustManager');
} catch (e) {
send('[frida] Hook failed: ' + e); });

```

**What this does:** when the app calls SSLContext.init, Frida replaces the trustManagers argument with one that accepts everything — so the app won't reject Burp's certificate.

### Practical tips & caveats (for beginners)

**Run early:** launch the app under Frida with -f <package> so hooks are installed before SSL initialization.

**Match versions:** Frida scripts depend on the app using standard Java classes — obfuscated or native TLS may bypass this.

**Check logs:** use adb logcat and Frida output to diagnose problems.

**Combine with CA:** often you both install Burp CA into system store **and** use Frida repin as fallback for reliability.

**Ethics & safety:** never intercept or publish live user tokens or PII. Use test accounts only and redact any sensitive values.

### One-line summary for your report

**SSLContext.init configures which trust managers the app uses for TLS; hooking it with Frida lets you inject a TrustManager that accepts the proxy's certificate (or accepts all), enabling MITM inspection of HTTPS traffic in a controlled lab.**

---

RootAVD.bat?

**ChatGPT said:**

Thought for 8s

Gotcha — rootAVD.bat is the Windows batch script from the *rootAVD* tool. It automates **rooting an Android Studio emulator (AVD)** by patching the AVD's boot/ramdisk so you can run **Magisk/root** inside the emulator.

**What it's for**

Get **root** on an emulator (for testing only).

Makes it easier to **install a system CA** (for MITM with Burp) and run tools that need root.

**How it's typically used (Windows)**

You run commands like:

rootAVD.bat listAllAVDs — list available AVDs.

rootAVD.bat <path-to-ramdisk.img> — patch the ramdisk for that system image.

Then you launch the AVD with the **patched ramdisk**. (Always keep a backup of the original ramdisk.img.)

**Requirements / gotchas**

Use a **Google APIs x86/x86\_64** system image that matches your AVD.

Close any running emulators before patching.

Make sure your **Android SDK path** is set; run the script from a location with write permissions (sometimes “Run as administrator” helps).

Rooted AVDs can be **unstable** and will **break Play Integrity/SafetyNet**—OK for labs, not for production.

**When to use it in your workflow**

Before installing the **Burp CA** to the **system** trust store (Magisk + RootAVD helps).

When Frida hooks or trust-store changes require **root** on the emulator.

rootAVD.bat?

ChatGPT said:

Thought for 8s

Gotcha — `rootAVD.bat` is the Windows batch script from the `rootAVD` tool. It automates **rooting an Android Studio emulator (AVD)** by patching the AVD's boot/ramdisk so you can run **Magisk/root** inside the emulator.

What it's for

Get **root** on an emulator (for testing only).

Makes it easier to **install a system CA** (for MITM with Burp) and run tools that need root.

How it's typically used (Windows)

You run commands like:

```
rootAVD.bat listAllAVDs — list available AVDs.
```

```
rootAVD.bat <path-to-ramdisk.img> — patch the ramdisk for that system image.
```

Then you launch the AVD with the **patched ramdisk**. (Always keep a backup of the original `ramdisk.img`.)

Requirements / gotchas

Use a **Google APIs x86/x86\_64** system image that matches your AVD.

Close any running emulators before patching.

Make sure your **Android SDK path** is set; run the script from a location with write permissions (sometimes “Run as administrator” helps).

Rooted AVDs can be **unstable** and will **break Play Integrity/SafetyNet**—OK for labs, not for production.

When to use it in your workflow

Before installing the **Burp CA** to the **system** trust store (Magisk + RootAVD helps).

When Frida hooks or trust-store changes require **root** on the emulator.

If you want, paste the exact command you plan to run with `rootAVD.bat` and your AVD name, and I'll sanity-check it.

You said:

Frida codeShare js

ChatGPT said:

Thought for 9s

Got it — **Frida CodeShare JS** refers to ready-made Frida JavaScript hooks that are published on Frida’s CodeShare and can be loaded directly from the CLI.

What it is

**CodeShare** = a catalog of community Frida scripts (JavaScript).

You can load a script **without downloading it**, using a special `codeshare`: URL.

How to use a CodeShare script (Android)

```
# Launch target app and load a CodeShare script  
frida -U -f com.your.app -l codeshare:AUTHOR/REPO
```

Examples you’ll see often for **SSL-pinning bypass** (names may vary):

`codeshare:akabel/frida-android-repinning`

`codeshare:pcipolloni/universal-android-ssl-pinning-bypass`

`codeshare:CapricornusC/okhttp3-unpinning (OkHttp3/4 pinning)`

Replace AUTHOR/REPO with the exact CodeShare slug.

If you want a local file instead

```
frida -U -f com.your.app -l /path/to/bypass_ssl.js
```

Minimal JS patterns (for reference)

These are typical hook points; exact classes may differ per ROM/library version.

```
Java.perform(() => {  
    // 1) TrustManagerImpl (AOSP Conscrypt)  
    try {  
        const TMI =  
        Java.use('com.android.org.conscrypt.TrustManagerImpl');  
  
        TMI.checkServerTrusted.overload('[Ljava.security.cert.X509Certificate;  
        , 'java.lang.String', 'java.lang.String').implementation =
```

```

        function(chain, authType, host) { return []; };
    } catch (e) {}

    // 2) OkHttp3 CertificatePinner
    try {
        const CertPinner = Java.use('okhttp3.CertificatePinner');
        CertPinner.check.overload('java.lang.String',
        'java.util.List').implementation =
            function(host, peerCerts) { return; };
    } catch (e) {}

    // 3) X509TrustManager (generic)
    try {
        const X509TM = Java.use('javax.net.ssl.X509TrustManager');
        // (interfaces are tricky; many scripts replace the instance
        instead)
    } catch (e) {}
});
```

## Troubleshooting tips

**Class not found** → The app may use **Play Services Conscrypt** (`com.google.android.gms.org.conscrypt`) or a **custom/obfuscated** trust manager. Search the app's classes for “pin”, “pinner”, or “TrustManager”.

**OkHttp v5 / custom pins** → Hook `okhttp3.CertificatePinner` (method signatures changed across versions).

**No traffic** → Ensure system-store **Burp CA** is installed (Magisk/RootAVD) and the app honors the proxy.

**Crashes after attach** → Try a different bypass script; attach *early* (`-f` to spawn), or delay hooks with `setTimeout`.

## Ethical reminder

Use these scripts **only** on apps/accounts you're authorized to test and on **lab devices/emulators**.

If you paste the exact CodeShare slug you're trying to load (e.g., `codeshare:akabel/frida-android-repinning`) and your app's HTTP stack (OkHttp? Volley? custom?), I'll tailor the exact `frida` command and, if needed, a small JS tweak for your case.

You said:

OKHttp3/4?

ChatGPT said:

Thought for 12s

Gotcha — you're asking about **OkHttp 3 vs 4** and what it means for Frida hooks.

The short version

**OkHttp 3.x** and **OkHttp 4.x** both live in the **same package**: `okhttp3.*`.

4.x is mostly **Kotlin**, but it's **API-compatible** with 3.x for the parts you care about.

`CertificatePinner` still exists and is the main SSL-pinning gate in both.

How to tell which one the app uses

Decompile (`jad`) and look for a file like `okhttp3/OkHttpClient.class` → check embedded version strings (sometimes present).

In traffic, some apps add `okhttp/4.x` in a header, but don't rely on it.

Practically: you don't need the exact version; write hooks that cover both signatures.

Frida hooks that work for both 3 & 4

```
Java.perform(() => {
    // 1) Kill CertificatePinner checks (3.x & 4.x)
    try {
        const CertPinner = Java.use('okhttp3.CertificatePinner');

        // Most common (OkHttp 3 & 4 public)
        try {
            CertPinner.check.overload('java.lang.String',
'java.util.List').implementation = function(host, peerCerts) {
                // bypass
                return;
            };
        } catch (e) {}

        // Some builds have an extra param (seen in forks/obfuscations)
        try {
            CertPinner.check.overload('java.lang.String', 'java.util.List',
'java.lang.String').implementation = function(h, l, p) {
                return;
            };
        } catch (e) {}}
```

```

// Kotlin synthetic method in some 4.x builds (name may differ)
// If you see something like check$okhttp in jadx, try this:
try {
    CertPinner['check$okhttp'].overload('java.lang.String',
'java.util.List').implementation = function(h, l) {
    return;
}
} catch (e) {}
} catch (e) {}

// 2) (Optional) Nuke hostname verification to avoid hostname
mismatches
try {
    const HostVerifier =
Java.use('okhttp3.internal.tls.OkHostnameVerifier');
    HostVerifier.verify.overload('java.lang.String',
'javax.net.ssl.SSLSession').implementation = function(host, sess) {
        return true;
}
} catch (e) {}
});

```

## Tips

If `check` isn't hit, the app might pin **outside** OkHttp (e.g., Conscrypt/Play Services) — also hook TrustManager (AOSP/Google Conscrypt) or use a universal pinning-bypass script.

Always **spawn** the app (`-f`) so hooks load **before** network init.

If the method name you see in jadx is `check$okhttp`, include that overload too (as above).

If you paste a specific class/method you see in jadx (e.g., `okhttp3.CertificatePinner.check$okhttp(String, List)`), I'll tailor the exact Frida overload for your build.

---

`SSLContext.init(...)`

### ChatGPT said:

Great target to hook. Here's what `SSLContext.init(...)` is and why it's useful in your Frida workflow.

### What `SSLContext.init(...)` does

SSLContext is the Java/Android TLS engine. Calling init(...) wires together:

**KeyManagers** – present *client* certs to servers (mutual TLS, rare in mobile apps).

**TrustManagers** – decide whether the *server* cert/chain is trusted (pinning lives here or upstream).

**SecureRandom** – PRNG used during handshakes.

### Common Java signature

```
void SSLContext.init(  
    KeyManager[] keyManagers,  
    TrustManager[] trustManagers,  
    SecureRandom secureRandom  
) throws KeyManagementException
```

If an app passes null for trustManagers, the JRE picks the **default** TrustManagers (usually from the system keystore / Conscrypt).

If it passes **its own** TrustManagers (or a pinned store), that's how app-level **certificate pinning/trust customization** happens.

### Why hook it

By intercepting init(...), you can **replace** the app's TrustManagers with **your own** (e.g., one that trusts your Burp CA or returns success), thereby defeating pinning that happens in Java space.

### Frida hook pattern

```
Java.perform(() => {  
    const SSLContext = Java.use('javax.net.ssl.SSLContext');  
  
    const initOv = SSLContext.init.overload(  
        '[Ljavax.net.ssl.KeyManager;',  
        '[Ljavax.net.ssl.TrustManager;',  
        'java.security.SecureRandom'  
    );  
  
    initOv.implementation = function (kms, tms, sr) {  
        // tms = app-supplied TrustManagers (possibly pinned)  
        // Replace with your own (tmf.getTrustManagers()) or a no-op TrustManager  
        console.log('[*] SSLContext.init called, swapping TrustManagers');  
        return initOv.call(this, kms, myTmf.getTrustManagers(), sr);  
    };  
});
```

Where myTmf is built from a KeyStore that contains **your CA** (as in your repinning script).

## Notes & pitfalls

**Multiple initializations:** Some apps create several SSLContexts (OkHttp, custom clients). Your hook runs each time; log and ensure your replacement occurs **before** network use (spawn the app with -f).

**Native / Conscrypt path:** If pinning happens **outside** Java (native libs / com.google.android.gms.org.conscrypt.TrustManagerImpl), SSLContext.init alone may not be enough—also hook TrustManagerImpl.checkServerTrusted(...) and OkHttp's CertificatePinner.

**Hostname mismatch:** Even with trusted chains, hostname verification can still fail. Hook okhttp3.internal.tls.OkHostnameVerifier.verify(...) or set a permissive HostnameVerifier when appropriate (lab only).

**Scope of trust:** Replacing TrustManagers globally trusts your CA for **all hosts** used by that SSLContext. For tighter control, you can:

Inspect the host later (e.g., in verify(...)) and allow only certain domains.

Build different SSLContexts per host (harder to control from outside).

## Quick “trust-all” vs “trust-my-CA”

**Trust-all (not recommended beyond labs):**

```
const X509TM = Java.registerClass({
    name: 'com.example.TrustAll',
    implements: [Java.use('javax.net.ssl.X509TrustManager')],
    methods: {
        checkClientTrusted: function() {},
        checkServerTrusted: function() {},
        getAcceptedIssuers: function() { return []; }
    }
}); // pass [new TrustAll()] as tms
```

**Trust-my-CA (safer for testing):** Use KeyStore + TrustManagerFactory (what your script does), so only chains under **your intercept CA** succeed.

## How to validate your hook is active

Log a line in the implementation and confirm it appears **before** first HTTPS request.

Intentionally **remove** your Burp CA and confirm handshakes start failing again (then restore) to sanity-check the path.

In Burp, verify the server cert chain shows **issued by your CA** for intercepted flows.

## When SSLContext.init won't be enough

App pins in **native code** (JNI/C++). You'll need to hook native functions (e.g., `SSL_set_custom_verify`, `X509_verify_cert`, or libcurl/boringssl entry points) — that's a different layer.

App uses **network security config** with pin sets and **certificate transparency** checks; combine with OkHttp pinner hooks and TrustManagerImpl overrides.

## 5. Work log table:

Date	Hours	Description of work
11-10-2025	4	Searched for suitable tools.
12-10-2025	2	Searched for mobile apps.
18-10-2025	3	Selected tools and apps for testing.
22-10-2025	5	Worked on the project proposal.
24-10-2025	1	Continued refining the project proposal.
28-10-2025	2	Searched APKMirror; installed Frida-server; installed Instagram and Telegram on the emulator; configured Android_35_GAPI environment for initial testing.
29-10-2025	2	Watched tutorial videos on Frida and SSL pinning bypass.
30-10-2025	4	Attempted Frida JS injection using CodeShare scripts; encountered multiple injection failures; debugged the error messages to identify the cause.
31-10-2025	7	Researched Burp Suite and proxy setup; tested proxy + Frida on Android_35_GAPI and Android_36_GAPI. Instagram frequently crashed after JS injection; troubleshooting was unsuccessful. Attempted Magisk setup (unsuccessful). Watched additional tutorial videos.
02-11-2025	8	Attempted to install AlwaysTrustUserCerts and Magisk on Android_36_GAPI; repeated Instagram injection process with multiple failures. Instagram remained unstable.
03-11-2025	4	Repeated the process for Telegram on Android_36_GAPI. Telegram failed to log in, likely due to proxy or certificate-trust configuration issues.
04-11-2025	5	Repeated the entire setup from scratch using Android_35_GAPI: installed a new Instagram APK and the same Frida-server version; achieved successful interception after multiple attempts. After re-login, Instagram became unresponsive, likely due to changed sessions/tokens, background services retaining old connections, or hooks/proxy changes that broke the TLS handshake and caused the UI to hang.
05-11-2025	8	Switched to Android_31_GAPI, installed Magisk using RootAVD, and added the Burp CA to the system store. Installed a new Instagram APK, used a new Frida bypass script (JS), and repeated the process; Frida injection succeeded after multiple trials.

<b>06-11-2025</b>	<b>6</b>	Captured, organized, and analyzed intercepted request and response traffic; extracted the first example of forensic data and studied JSON.
<b>07-11-2025</b>	<b>6</b>	Analyzed additional captured data; documented the second and third example outputs; began writing detailed results and the JSON structure.
<b>08-11-2025</b>	<b>3</b>	Continued refining the analysis and writing of the report for the three examples.
<b>09-11-2025</b>	<b>4</b>	Started intercepting Telegram; the attempt was unsuccessful.
<b>13-11-2025</b>	<b>2</b>	Worked on completing the Progress Report Template.

**Total Hours : 76**