

Final REPORT

CSIS4440-003

Group No. 01

Prepared By

Hind Abu Alamrain
Shiksha Kaharanthilake

October 24, 2025

Vedio Link:

https://1drv.ms/v/c/6302570498dd3f82/IQB_FT-NPltqRLluKScrSiMAQGgW37956DtHC9hvSEyPlk?e=1dkHsY

Student 1: Team Leader

Title: Runtime Network Evidence Extractor

Student Name: Hind Abu Alamrain

Student ID: 300403234

Abstract:

This project uses **Frida** to bypass **SSL pinning** and **Burp Suite** to intercept and analyze Instagram's and Tusky's (for Mastodon) HTTPS traffic on Android. The captured data reveals hidden API endpoints, device identifiers, session tokens, navigation patterns, and other runtime artifacts. The goal is to understand app behavior, extract key forensic evidence, and demonstrate a repeatable analysis workflow.

Background Research:

Mobile applications communicate with backend servers using HTTPS, often hiding important runtime information behind TLS encryption and certificate pinning. Runtime instrumentation tools like **Frida** allow dynamic analysis by injecting custom scripts to intercept function calls, manipulate runtime behavior, and bypass security mechanisms. This project uses **Frida for dynamic instrumentation** and **Burp Suite as a MITM proxy** to capture and analyze real-time network traffic from Android apps. Key background areas researched include:

- Android certificate trust stores (system vs. user CA).
- SSL/TLS interception and why applications reject user-installed certificates.
- Certificate pinning, TLS handshake validation, and how apps detect interception attempts.
- The need for rooted devices, Magisk-patched images, or system-level CA installation.
- MITM techniques, Frida-based SSL pinning bypass scripts, and common bypass failures.

All testing was conducted on controlled Android emulators and authorized test accounts.

Purpose

The objective of this project is to extract **runtime network evidence** from mobile applications for forensic analysis. By combining Frida-based instrumentation with Burp Suite MITM interception, the workflow captures:

- Hidden API endpoints.
- Device identifiers and session tokens.
- Navigation flows and action sequences.
- Timestamps linking user actions to backend events.
- Authenticated account details returned by internal APIs.

This forensic evidence allows investigators to reconstruct user timelines, validate account activity, and identify key artifacts that would otherwise remain encrypted during normal application use.

Forensic Plan

1. How the Tools Work

Frida is a dynamic instrumentation toolkit that injects JavaScript into running Android processes. It allows the examiner to hook functions, modify runtime behavior, and bypass security mechanisms such as SSL pinning.

Burp Suite functions as a MITM (man-in-the-middle) proxy that intercepts HTTPS traffic once the device trusts its certificate. By routing all emulator traffic through Burp, the examiner can capture API requests and responses from mobile applications in real time.

2. How the Examination Will Be Conducted

Prepare the Environment

- Launch a rooted Android emulator (Android 31).
- Install the Frida-server version that matches the device ABI.
- Install the target applications (Instagram and Tusky).

Enable MITM Interception

- Configure emulator-wide proxy settings to route traffic to Burp Suite.
- Install Burp's CA certificate into the **system** trust store using Magisk and RootAVD.

Bypass SSL Pinning

Load a Frida SSL-bypass script (`bypass_ssl.js`) to intercept SSL functions, disable certificate checks, and enable HTTPS traffic interception.

Perform the Analysis

- Launch the target app with Frida injection.
- Use the application normally (e.g., login, open feeds).
- Collect HTTPS requests and responses in Burp Suite.
- Extract key forensic artifacts such as identifiers, session tokens, endpoints, timestamps, and account details.

Document Findings

- Convert captured traffic into structured JSON records.
- Capture screenshots of relevant requests and responses.
- Record failures, crashes, or anti-instrumentation behaviors (e.g., TLS errors, app detection of Frida).

3. Underlying Implementation / Architecture

Rooted Emulator Architecture

The examination uses RootAVD to generate a modified Android emulator image that supports system-level modifications. Magisk is integrated to allow system-wide CA installation. This enables the device to trust Burp Suite's certificate and permits HTTPS interception.

Frida Instrumentation Architecture

Frida-server runs inside the emulator at `/data/local/tmp`.

The examiner connects to it using `frida-tools` on the host machine.

Frida hooks Java and native SSL functions at runtime, preventing certificate validation and allowing MITM inspection.

Traffic Routing Architecture

All emulator traffic is routed through `127.0.0.1:8080` (Burp listener).

Burp Suite decrypts, logs, and displays API traffic after the Frida SSL-bypass hook ensures the app accepts the intercepted certificates.

Detailed Analysis:

This section documents the complete hands-on workflow used to intercept and analyze Instagram's runtime HTTPS traffic. Commands, screenshots, and observed API responses are included as required by the assignment.

1. Start the rooted AVD

Command:

emulator -avd Android_31

The rooted Android 31 emulator is launched to allow system-level CA installation and Frida instrumentation.

2. Install Python 3 and Frida-tools

Commands:

pip3 install frida frida-tools

These tools provide the Frida client needed to attach to processes inside the emulator.

3. Check device CPU ABI

Command:

adb shell getprop ro.product.cpu.abi

This determines the correct Frida-server binary for the emulator architecture.

4. Install Instagram APK

Command:

Instagram: adb install instagram-v361.0.0.46.88-x86_64.apk

Tusky: adb install Tusky.apk

After installation, Instagram appears on the emulator and is ready for instrumentation.

5. Push and run Frida-server

Commands:

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

This spawns Frida-server inside the emulator, allowing the host machine to inject scripts.

6. Verify Frida can see Instagram

Command:

```
frida-ps -U
```

This confirms that Frida-server is running correctly and can list Android processes.

7. Configure Burp Suite Proxy

Command:

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

All network traffic is now routed through the Burp proxy.

8. Push Burp Suite CA Certificate

Command:

```
adb push burp.cer /sdcard/
```

9. Install CA into system trust store

Command:

```
cd rootAVD-master\rootAVD-master  
rootAVD.bat listAllAVDs  
rootAVD.bat system-images\android-31\google_apis\x86_64\ramdisk.img  
emulator -avd Android_31  
cd AppData\Local\Android\Sdk\platform-tools  
adb push AlwaysTrustUserCerts_v1.3 /sdcard
```

RootAVD + Magisk enable system-level CA installation, allowing Burp to decrypt HTTPS traffic.

10. Run Frida SSL-pinning bypass and spawn Instagram

Command:

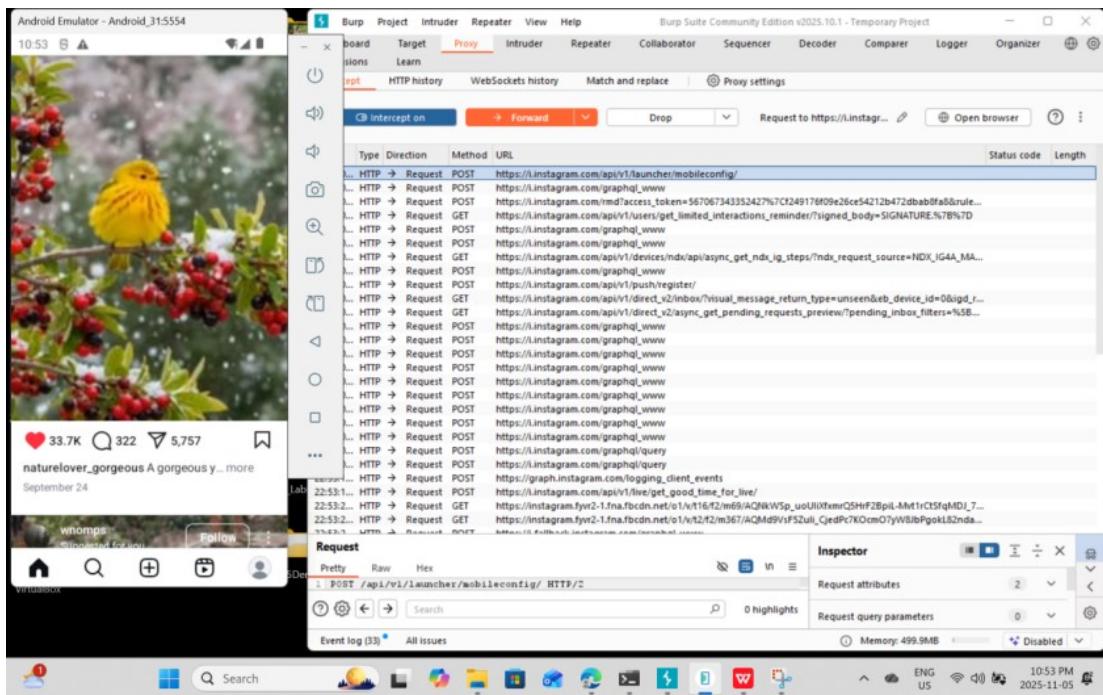
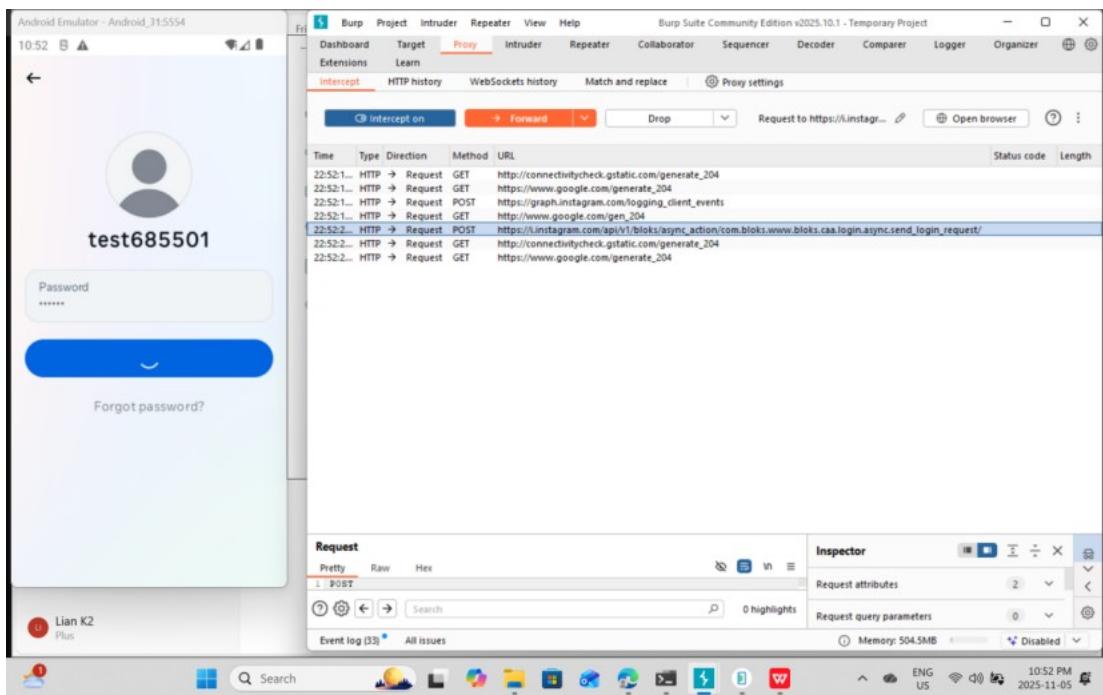
Instagram: frida -U -f com.instagram.android -l bypass_ssl.js

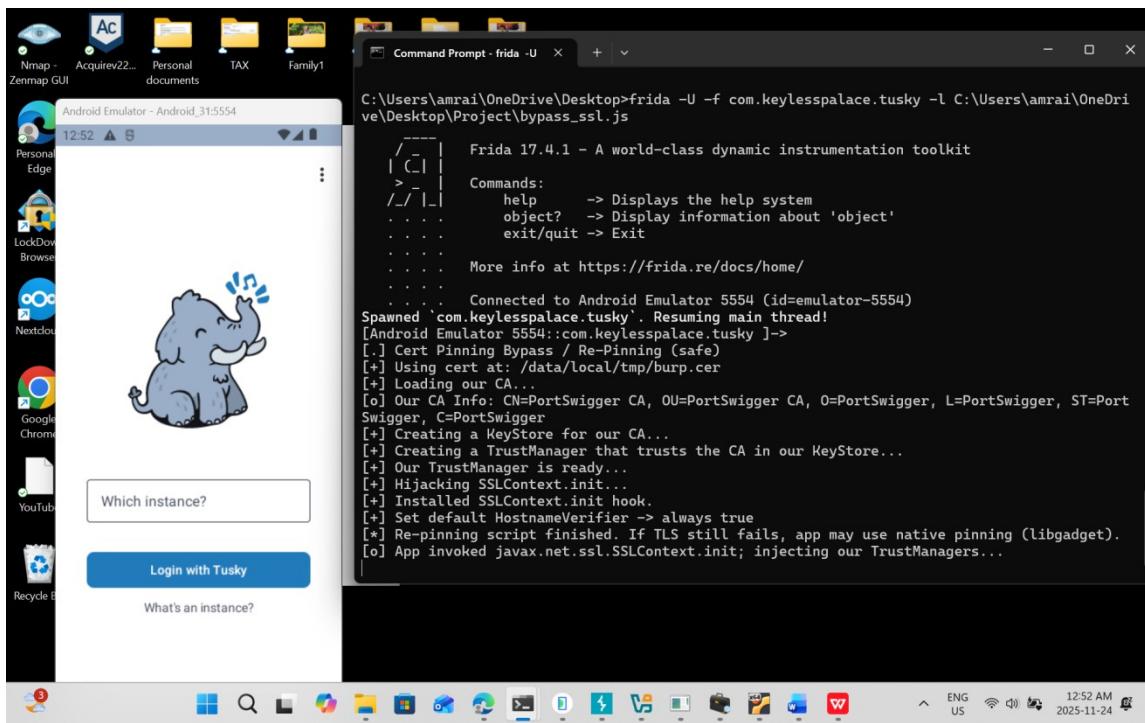
Tusky: frida -U -f com.keylesspalace.tusky -l bypass_ssl.js

Instagram launches with SSL pinning disabled.

11. Monitor Instagram traffic in Burp Suite

Instagram's HTTPS requests become visible.





Example captured endpoints:

1. Instagram:

Example1: login Request:

Endpoint: /api/v1/bloks/apps/com.bloks.www.caa.login.login_homepage

Key forensic artifacts extracted:

Status Code: HTTP/2 200 OK

Server timestamp

Device identifiers: X-Ig-Device-Id , X-Ig-Family-Device-Id , X-Ig-Android-Id

Session identifiers: X-Pigeon-Session-Id, X-Pigeon-Rawclienttime (client timestamp)

Navigation metadata: X-Ig-Nav-Chain

Network context : Connection type (WIFI)

User-Agent: Instagram 361.0.0.46.88 Android 31

Interpretation:

This request confirms the start of the Instagram login workflow and ties specific device IDs and session tokens to the user action of opening the login page.

POST request to https://i.instagram.com/api/v1/bloks/apps/com.bloks.www.caa.login.login_homepage/

Request	Response			
Pretty	Raw	Hex		
<pre> 1 POST /api/v1/bloks/apps/com.bloks.www.caa.login.login_homepage/ HTTP/2 2 Host: i.instagram.com 3 X-Ig-App-Locale: en_US 4 X-Ig-Device-Locale: en_US 5 X-Ig-Mapped-Locale: en_US 6 X-Pigeon-Session-Id: [REDACTED] 7 X-Pigeon-Rawclientime: 1762405550.740 8 X-Ig-Bandwidth-Speed-Kbps: 2930.000 9 X-Ig-Bandwidth-Totalbytes-B: 0 10 X-Ig-Bandwidth-Totaltime-Ms: 0 11 X-Ig-Block-Version-Id: 16e5197b5c8710eaafdf803935ed8c450a1a2e3eb656bfff104df088b500bcf 12 X-Ig-BlockVersion 13 X-Bloks-Prism-Button-Version: CONTROL 14 X-Bloks-Prism-Colors-Enabled: false 15 X-Bloks-Prism-Az-Base-Colors-Enabled: false 16 X-Bloks-Prism-Font-Enabled: false 17 X-Bloks-Is-Layout-rtl: false 18 X-Ig-Device-Id: [REDACTED] 19 X-Ig-Family-Device-Id: [REDACTED] - web - 202000 20 X-Ig-Android-ID: [REDACTED] - web - 202000 21 X-Ig-Timezone-Offset: -20800 22 X-Ig-Nav-Chain 23 X-FB-Connection-Type: WIFI 24 X-Ig-Connection-Type: WIFI 25 X-Ig-Capabilities: 3bRv10+ 26 X-Ig-App-Id: 567067343352427 27 Priority: u+3 28 User-Agent: Instagram 361.0.0.46.88 Android (31/1; 160dpi; 320x640; Google/google; sdk_gphone64_x86_64; emulator64_x86_64_arm64; ranchu; en_US; [REDACTED]) 29 Accept-Language: en-US 30 X-HM-AppId: 567067343352427 31 X-Intended-User-Id: 0 32 Content-Type: application/x-www-form-urlencoded; charset=UTF-8 33 Content-Length: 1170 34 Accept-Encoding: gzip, deflate, br 35 X-FB-Http-Engine: Liger 36 X-FB-Client-IP: True 37 X-FB-Server-Cluster: True </pre>				
?	⚙️	⟳	Search	0 highlights

Windows Taskbar:

System Tray:

POST request to https://i.instagram.com/api/v1/bloks/apps/com.bloks.www.caa.login.login_homepage/

Request	Response			
Pretty	Raw	Hex	Render	
<pre> 1 HTTP/2 200 OK 2 Content-Type: application/json; charset=utf-8 3 Date: Thu, 06 Nov 2025 05:12:31 GMT 4 Vary: Accept-Language, Cookie, Accept-Encoding 5 Content-Language: en 6 Strict-Transport-Security: max-age=31536000 7 Cache-Control: private, no-cache, no-store, must-revalidate 8 Pragma: no-cache 9 Expires: Sat, 01 Jan 2000 00:00:00 GMT 10 X-FRAME-OPTIONS: SAMEORIGIN 11 Content-Security-Policy: report-uri https://www.instagram.com/security/csp_report/; default-src 'self' https://www.instagram.com; img-src data: blob: https://*.fbcdn.net https://*.fbcdn.net https://*.instagr.am https://*.facebook.com https://*.fbabs.com https://*.threads.com https://*.giphy.com font-src data: https://*.fbcdn.net manifest-src 'self' https://www.instagram.com; media-src 'self' blob: https://www.instagram.com https://*.cdninstagram.com; script-src 'self' https://instagr.am https://www.instagram.com https://*.cdninstagram.com; style-src 'self' https://www.instagram.com https://*.instagr.am https://*.fbcdn.net unsafe-inline' unsafe-eval' blob; style-src 'self' https://graph.instagram.com https://*.graph.instagram.com https://i.instagram.com https://*.i.instagram.com https://*.od.threads.net https://*.od.threads.net https://*.od.threads.net https://www.instagram.com https://edge-chat.instagram.com https://*.facebook.com https://*.fbcdn.net https://*.fbcdn.net chrome-extension://badgeojelhgndahlyhdicfhalfpadf blob; worker-src 'self' blob: https://www.instagram.com frame-src 'self' https://instagr.am https://www.instagram.com https://staticxx.facebook.com https://www.facebook.com https://web.facebook.com https://connect.facebook.net https://m.facebook.com https://*.fbabs.com object-src 'none'; upgrade-insecure-requests 12 Cross-Origin-Embedder-Policy-Report-Only: require-corp-report-to="coop" 13 Report-To: {"group": "coop", "max_age": 86400, "endpoints": [{"url": "/security/corp_report/"}]}, {"group": "coop", "max_age": 86400, "endpoints": [{"url": "/security/report/"}]} 14 Cross-Origin-Bridge-Policy: cross-origin 15 Cross-Origin-Open-Policy: same-origin-allow-popups;report-to="coop" 16 X-Content-Type-Options: nosniff 17 X-Yes-Protection: S 18 X-Ig-Push-State: c2 19 X-Ig-Cache-Control: cache 20 X-Aes: 624 21 X-Ig-Request-Elapsed-Time-Ms: 172 22 X-Ig-Peak-V2: 0 23 X-Ig-Peak-Time: 1 24 X-Ig-Capacity-Level: [] 25 X-Stack: distillery 26 Ig-Set-Ig-U-IG-Direct-Region-Hint: 27 Ig-Set-Ig-U-Shbid: 28 Ig-Set-Ig-U-Shbts: 29 Ig-Set-Ig-U-Rut: 30 Ig-Set-Ig-U-UDs-User-Id: 31 X-Data-Size: 5019914 55937 18000 </pre>				
?	⚙️	⟳	Search	0 highlights

Windows Taskbar:

System Tray:

Structured JSON Representation

The extracted data was converted into a simplified **forensic JSON record**, highlighting only the critical fields necessary for analysis.

```

{
  "request_id": "req-0001",
  "session_id": "<REDACTED_SESSION_ID>",
  "meta": {
    "app": "com.instagram.android",
    "capture_tool": "frida",
    "script_name": "bypass-ssl.js"
  },
  "ts_request": "2025-11-06T05:12:30.740Z",
  "ts_response": "2025-11-06T05:12:31.000Z",
  "url": "https://i.instagram.com/api/v1/blocks/apps/com.bloks.www.caa.login.login_homepage/",
  "method": "POST",
  "headers": {
    "X-Ig-Device-ID": "<REDACTED_DEVICE_ID>",
    "X-Ig-Family-Device-ID": "<REDACTED_DEVICE_ID>",
    "X-Ig-Android-ID": "<REDACTED_ANDROID_ID>",
    "X-Pigeon-Session-ID": "<REDACTED SESSION ID>",
    "X-Pigeon-RawClienttime": "1762405950.740",
    "User-Agent": "Instagram 361.0.0.46.88 Android (31/12; 160dpi; 320x640; Google/google; emulator64_x86_64_arm64; en_US)",
    "X-Fi-Connection-Type": "WIFI",
    "X-Ig-Connection-Type": "WIFI"
  },
  "response": {
    "status_code": 200,
    "headers": [
      {
        "Date": "Thu, 06 Nov 2025 05:12:31 GMT",
        "Content-Type": "application/json; charset=utf-8",
        "Strict-Transport-Security": "max-age=31536000",
        "X-Frame-Options": "SAMEORIGIN",
        "X-Ig-Origin-Region": "<REDACTED_REGION>"
      }
    ],
    "artifacts": {
      "endpoint": [
        "https://i.instagram.com/api/v1/blocks/apps/com.bloks.www.caa.login.login_homepage"
      ],
      "device_id": {
        "device_id": "<REDACTED_DEVICE_ID>",
        "android_id": "<REDACTED_ANDROID_ID>",
        "pigeon_session_id": "<REDACTED_SESSION_ID>" }
    },
    "notes": "Captured via Frida (bypass-ssl.js) on emulator; no request body present for this request."
  }
}

```

Example 2 : Account Family Check (“Who Logged In”):

Endpoint: GET /api/v1/multiple_accounts/get_account_family/

Key forensic artifacts extracted:

Status code: 200

Client & Server timestamps

Authenticated account: username, full_name, pk / id (unique numerical identifier)

Session tracker: session_id

Session status: status: ok

Routing region: X-Ig-Origin-Region:

Authorization: Bearer

Interpretation:

This provides **proof of successful authentication** and establishes exactly *which account* was active on the device at that time.

GET request to https://b.i.instagram.com/api/v1/multiple_accounts/get_account_family/?request_source=com.bloks.www.caa.login.login_homepage

Request Response

Pretty Raw Hex

```
1 GET /api/v1/multiple_accounts/get_account_family/?request_source=com.bloks.www.caa.login.login_homepage HTTP/2
2 Host: b.i.instagram.com
3 X-Ig-App-Locale: en_US
4 X-Ig-Device-Locale: en_US
5 X-Ig-Mapped-Locale: en_US
6 X-Ig-Region-Emailclient: [REDACTED]
7 X-Ig-Emailclient-Used-Https: -1.000
8 X-Ig-Bandwidth-Totalabytes-B: 0
9 X-Ig-Bandwidth-Totalbytes-Mb: 0
10 X-Xr-Session-Id: 16c9157b29710eafdf803935edc450a1a2e3eb69bfff104df008b900bcf
11 X-Ig-Www-Claim: hmac:AZ1FOLsXyYjTVEqjS1bR850v5mpjiBbbEHnZ-2_QcqY0nBvQ
12 X-Bloks-Prism-Button-Version: CONTROL
13 X-Bloks-Prism-Colors-Enabled: false
14 X-Bloks-Prism-Ax-Base-Colors-Enabled: false
15 X-Bloks-Prism-Ax-Font-Enabled: false
16 X-Bloks-Is-Layout-Rtl: false
17 X-Ig-Device-Id: [REDACTED]
18 X-Ig-Family-Device-Id: [REDACTED]
19 X-Ig-Android-Id: [REDACTED]
20 X-Ig-Timeszone-Offset: -08000
21 X-Ig-Nav-Chain: [REDACTED]
22 X-Ig-Connection-Type: WIFI
23 X-Ig-Location-Type: WIFI
24 X-Ig-Capabilities: 3hrTv10*
25 X-Ig-App-Id: 6706734335427
26 Priority: user3
27 X-Tigon-Is-Retry: True
28 X-Tigon-Is-Retry: True
29 X-Tigon-Is-Retry: True
30 X-Tigon-Is-Retry: True
31 User-Agent: Instagram/361.0.0.46.88 Android (31/12; 160dpi; 320x640; Google/google; adR_gphone64_x86_64; emulator64_x86_64_arm64; ranchu; en_US; [REDACTED])
32 Accept-Language: en-US
33 Authorization: Bearer eyJkcl8icWYcK1lkIjo1Nsc4NtrW0dc5MgslLCjZXNsawQ8uWQioi3Hsg10TA4Nsks0CuNQWd3cnR0bXASUGYimHZ1JTNBHTA1MOFBWhBYjBjTiwRghGaGQz0URDd0wUUB5Vd0tMuVdi1Bd3U5Um5TRHcifq==
34 X-Add-Header: A0dPAAABAHJlfuUS8sWYjeypkq#
35 Ig-U-Us-User-Id: 77055007938
36 Ig-U-Rur: NNA
37 X-Intended-User-Id: 77055007938
38 Accept-Encoding: gzip, deflate, br
39 X-Fb-Http-Engine: Liger
40 X-Fb-Client-Ip: True
41 X-Fb-Server-Cluster: True
42
```

GET request to https://b.i.instagram.com/api/v1/multiple_accounts/get_account_family/?request_source=caa.login.login_homepage

Request Response

Request	Response
Pretty	Raw
Hex	Render
1 HTTP/1.0 200 OK	
2 Content-Type: application/json; charset=utf-8	
3 Date: Sat, 08 Nov 2015 01:34:35 GMT	
4 Vary: Accept-Language, Cookie, Accept-Encoding	
5 Content-Language: en	
6 Strict-Transport-Security: max-age=31536000	
7 Cache-Control: private, no-cache, no-store, must-revalidate	
8 Pragma: no-cache	
9 Expires: Sat, 01 Jan 2000 00:00:00 GMT	
10 X-Framename: SAMEORIGIN	
11 Content-Security-Policy: report-uri https://www.instagram.com/security/csp_report/; default-src 'self' https://www.instagram.com; navigate-to 'none'; object-src 'none'; form-action 'none'; base-uri 'none'; upgrade-insecure-requests; sandbox; img-src https://*.giphy.com	
12 Cross-Origin-Embedder-Policy-Report-Only: require-corp;report-to=coop"	
13 Report-To: {"group": "coop", "max_age": 86400, "endpoints": [{"url": "/security/coop_report/"}}], {"group": "coop", "max_age": 86400, "endpoints": [{"url": "/security/coop_report/"}}]	
14 Cross-Origin-Opener-Policy: same-origin-allow-popups;report-to=coop"	
15 X-Content-Type-Options: nosniff	
16 X-Xss-Protection: 0	
17 X-Ig-Push-State: c2	
18 X-Ig-Cache-Control: cache	
19 X-Ad: E2	
20 X-Ig-Request-Elapsed-Time-Ms: 21	
21 X-Ig-Peak-V2: 0	
22 X-Ig-Peak-Time: 0	
23 X-Ig-Capacity-Level: 2	
24 X-Stack: distillery	
25 Ig-Set-Ig-U-Ig-Direct-Region-Hint:	
26 Ig-Set-Ig-U-Shbid:	
27 Ig-Set-Ig-U-Shbts:	
28 Ig-Set-Ig-U-Rur: 77859007938,1754101679:01fe1f3f07d0b1034+cf56+3329dddc56b3c7d3ddad1ccac08006f9fc930cb6ca700	
29 Ig-Set-Ig-U-Ds-User-Id: 77859007938	
30 X-Perf-Stats: IS20602215656_22130	
31 Content-Length: 954	
32 X-Ig-Origin-Region: [REDACTED]	
33 X-Ib-Client-IP-Forwarded: [REDACTED]	
34 X-Ib-Server-Cluster-Forwarded: [REDACTED]	
35 Alt-Svc: h3-2.0; ma=60400	
36 X-Fb-Connection-Quality: EXCELLENT; q=0.9, rtt=10, rtx=0, c=97, mss=1392, tbw=173431, tpx=1, tpl=-1, uplat=78, ullat=0	
37	
38 {"child_accounts": [], "main_accounts": [{"current_account": {"pk_id": "77859007938", "full_name": "test", "strong_id__": "77859007938", "has_password": 1, "id": "77859007938"}, "username": "test05501", "is_private": false, "is_verified": false, "profile_pic_url": "https://instagram.fres22.fbcn.net/v/c51.2085-15/573323465_i12982544632307212_7278521664109726286_n.png?tp=xtp-jpg_o_+150x150_tt4_w0026e_fg=eyle22zbhjCBLX3h2y16InByhczmBtGtCg1LmRq7Wsbny4kTAuYs1lIqIw0026_nc.htm=https://instagram.fres22.fbcn.net/w0026_nc.ca=r100.w0026_nc.oc=06c229f7fb5616fJ1PqlotGYtFV21L4Kxe1ZaBYtCtPua0-w_50UfveBK28-0B.html"}]}	
0 highlights	

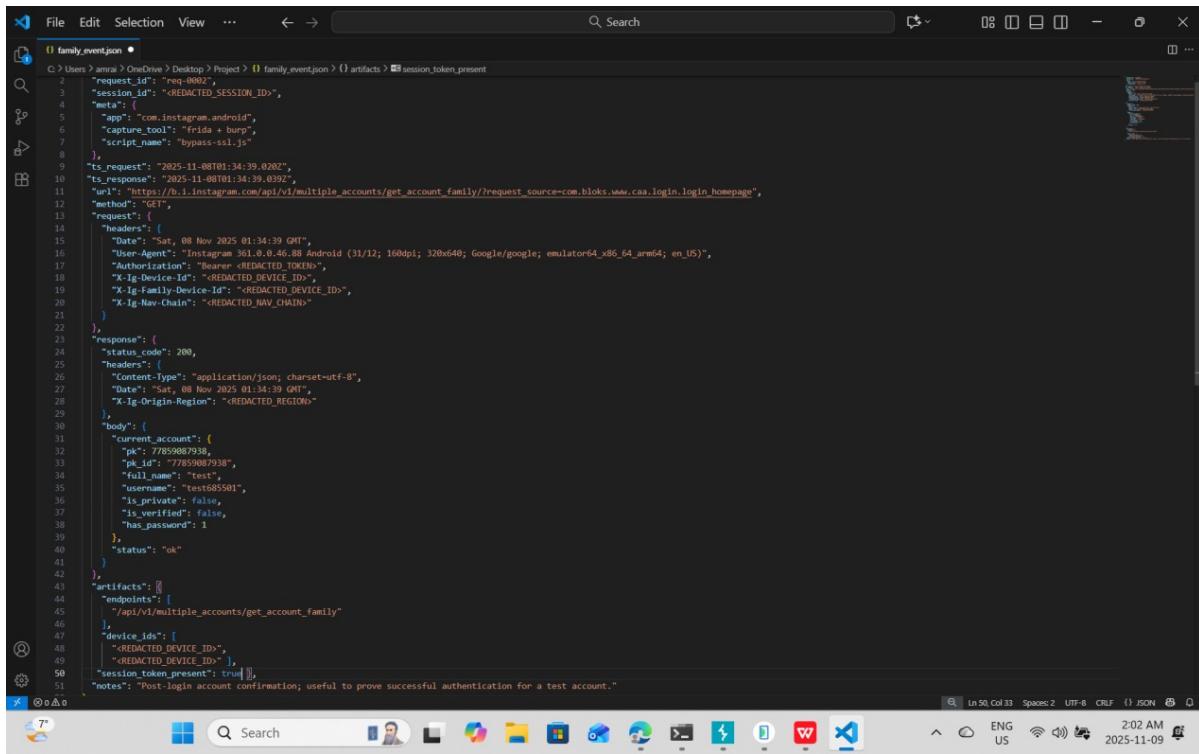
Search

ENG US

614 PM 2025-11-07

Structured JSON Representation

The extracted data was converted into a simplified **forensic JSON record**, highlighting only the critical fields necessary for analysis.



```
family.event.json
C:\Users\amrai\OneDrive\Desktop\Project\01\family.event.json\01\artifacts> session_token_present
1
2   "request_id": "red-0002",
3   "session_id": "<REDACTED_SESSION_ID>",
4   "media": [
5     {
6       "app": "com.instagram.android",
7       "capture_tool": "frida + burp",
8       "script_name": "bypass-sel.js"
9     },
10    "ts_request": "2025-11-08T01:34:39.020Z",
11    "ts_response": "2025-11-08T01:34:39.039Z",
12    "url": "https://0.i.instagram.com/api/v1/multiple_accounts/get_account_family/?request_source=com.bloks.www.ca.login.login_homepage",
13    "method": "GET",
14    "request": {
15      "headers": {
16        "Date": "Sat, 08 Nov 2025 01:34:39 GMT",
17        "User-Agent": "Instagram 361.0.0.46.88 Android (31/12; 160dpi; 320x640; Google/google; emulator64_x86_64_arm64; en_US)",
18        "Authorization": "Bearer <REDACTED_TOKEN>",
19        "X-Ig-Device-Id": "<REDACTED_DEVICE_ID>",
20        "X-Ig-Family-Device-Id": "<REDACTED_DEVICE_ID>",
21        "X-Ig-Nav-Chain": "<REDACTED_NAV_CHAIN>"
22      },
23      "response": {
24        "status_code": 200,
25        "headers": {
26          "Content-Type": "application/json; charset=utf-8",
27          "Date": "Sat, 08 Nov 2025 01:34:39 GMT",
28          "X-Ig-Origin-Region": "<REDACTED_REGION>"
29        },
30        "body": {
31          "current_account": {
32            "pk": 7759087938,
33            "pk_id": "7759087938",
34            "full_name": "test",
35            "username": "test88501",
36            "is_private": false,
37            "is_verified": false,
38            "has_password": 1
39          },
40          "status": "ok"
41        }
42      },
43      "artifacts": [],
44      "endpoints": [
45        "/api/v1/multiple_accounts/get_account_family"
46      ],
47      "device_ids": [
48        "<REDACTED_DEVICE_ID>",
49        "<REDACTED_DEVICE_ID>"],
50      "session_token_present": true
51    }
52  }
53  "notes": "Post-login account confirmation; useful to prove successful authentication for a test account."
```

Example 3: Post Comment Request:

Endpoint:

POST /api/v1/media/3764059087978684228/comment/

Key forensic artifacts extracted:

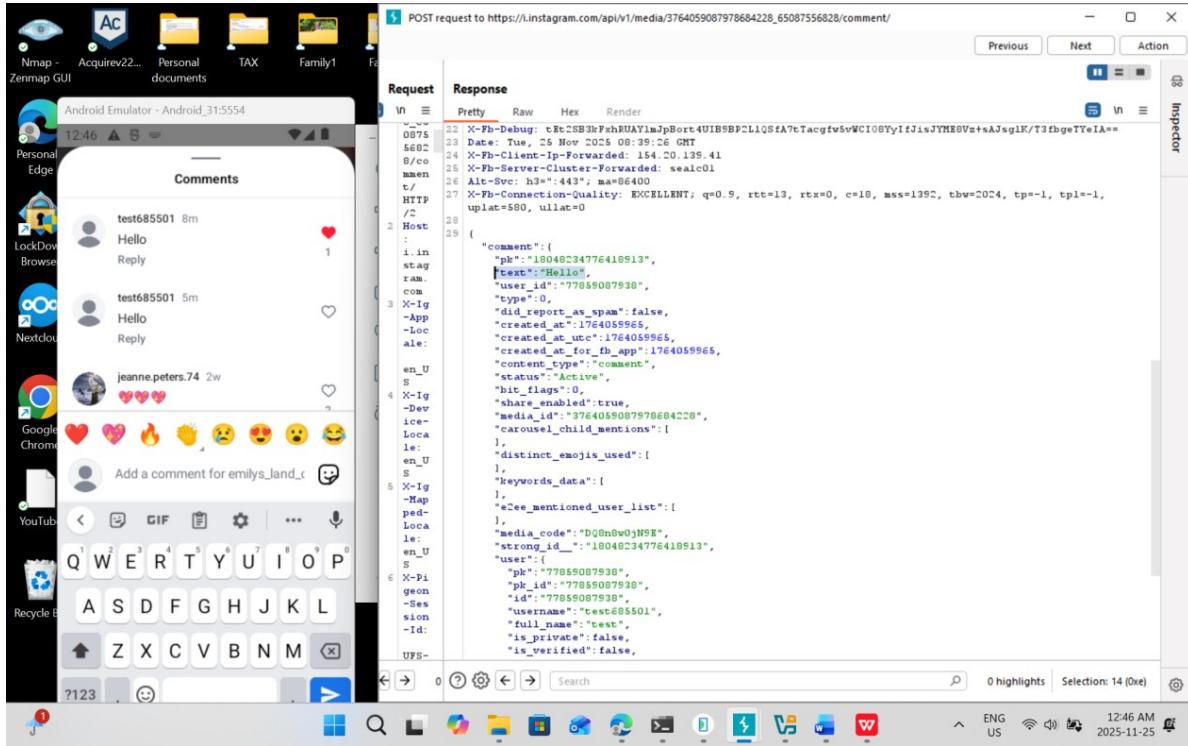
Status Code: HTTP/2 200 OK

Host: i.instagram.com

comment.text: "Hello"

Interpretation:

The text: "Hello" field shows the exact comment content the user submitted; its presence in the successful server response confirms that this message was posted from this account, on that device.



2. Tusky for Mastodon:

Example 1: App Registration Request

Endpoint: POST /api/v1/apps

Key forensic artifacts extracted:

Host: mastodon.social

App registration fields: client_name, redirect_uris, website

User Agent: Tusky (Android client)

Device context: Requests routed through emulator with Frida + Burp interception

Content-Type: application/x-www-form-urlencoded

OAuth fields (response): id - name - client_id - client_secret - vapid_key

Interpretation:

This request shows Tusky registering itself as an OAuth client with the Mastodon server. It confirms that HTTPS interception was successful and reveals how Tusky requests its permissions (scopes), identifies itself, and obtains OAuth credentials required for login.

The screenshot shows the Burp Suite interface with the following details:

HTTP history tab selected, showing the following log entries:

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TLS	IP	Cookies	Time
80	https://mastodon.social	GET	/api/v1/accounts/verify_credentials			200	1990	JSON			✓	151.101.193.91		00:31:	
81	https://mastodon.social	GET	/api/v1/accounts/verify_credentials			200	1990	JSON			✓	151.101.193.91		00:31:	
2920	https://mastodon.social	POST	/api/v1/apps		✓	200	1410	JSON			✓	151.101.65.91		01:19:	
2921	https://mastodon.social	GET	/api/v2/instance			200	25122	JSON			✓	151.101.65.91		01:19:	
2923	https://mastodon.social	GET	/auth/sign_in			200	18493	HTML		Log in - Mastodon	✓	151.101.65.91	_mastodon_sess...	01:19:	
2917	http://connectivitycheck.gst...	GET	/generate_204			204	127					142.251.33.67		01:19:	

Request pane (Pretty view):

```
1 POST /api/v1/apps HTTP/1.1
2 Host: mastodon.social
3 User-Agent: Tusky/29.0 Android/12 OkHttp/4.12.0
4 Content-Type: application/x-www-form-urlencoded
5 Content-Length: 149
6 Accept-Encoding: gzip, deflate, br
7 Connection: keep-alive
8
9 client_name=Tusky&redirect_uris=oAuth2Redirect%3A%2F%2Fcom.keylesspalace.tusky%2F
&scopes=read+write+follow+push&website=https%3A%2F%2Ftusky.app
```

Response pane (Pretty view):

```
10 X-Lache-nits: u, u, u
11 X-Timer: S1763975952.473478,V80,VE22S
12 Vary: Authorization, Origin, Accept-Encoding
13 Strict-Transport-Security: max-age=31557600
14 Alt-Svc: h3=":443";ma=86400,h3-27=:443;ma=86400
15
16 {
17   "id": "7037035",
18   "name": "Tusky",
19   "website": "https://tusky.app",
20   "scopes": [
21     "read",
22     "write",
23     "follow",
24     "push"
25   ],
26   "redirect_uris": [
27     "oAuth2Redirect://com.keylesspalace.tusky/"
28   ],
29   "vapid_key": "REDACTED"
30 }
31
32 {
33   "redirect_uri": "oAuth2Redirect://com.keylesspalace.tusky/",
34   "client_id": "REDACTED",
35   "client_secret": "REDACTED",
36   "client_secret_expires_at": 0
37 }
```

Bottom status bar: Memory: 324.5MB, Disabled, ENG US, 123 AM, 2025-11-24

Structured JSON Representation

The extracted data was converted into a simplified **forensic JSON record**, highlighting only the critical fields necessary for analysis.

The screenshot shows a code editor window with a dark theme. The file being edited is named 'Tusky_AppRegistration_.json'. The content of the file is a JSON object with the following structure:

```
1  {
2    "endpoint": "/api/v1/apps",
3    "host": "mastodon.social",
4    "method": "POST",
5
6    "request": {
7      "client_name": "Tusky",
8      "redirect_uris": "oauth://com.keylesspalace.tusky/",
9      "scopes": "read write follow",
10     "website": "https://tusky.app",
11     "user_agent": "tusky/29.0 (Android 12; OkHttp/4.12.0)"
12   },
13
14   "response": {
15     "id": "7837035",
16     "name": "Tusky",
17     "scopes": [
18       "read",
19       "write",
20       "follow"
21     ],
22     "redirect_uris": [
23       "oauth://com.keylesspalace.tusky/"
24     ],
25     "vapid_key": "<REDACTED>",
26     "client_id": "<REDACTED>",
27     "client_secret": "<REDACTED>"
28   }
29 }
30
```

The status bar at the bottom of the editor shows the following information: Ln 30, Col 1, Spaces: 4, UTF-8, CRLF, [] JSON, 2:15 AM, ENG US, 2025-11-24.

Example 2: Account Verification (“Who Logged In”)

Endpoint: /api/v1/accounts/verify_credentials

Host: mastodon.social

Key forensic artifacts extracted:

Status Code: HTTP/2 200 OK

User-Agent: tusky/29.0 (Android 12; OkHttp/4.12.0)

Authorization: Bearer <token>

Authenticated account details: id, username, acct, display_name

Timestamp: created_at

Profile media URLs: avatar , header

Server routing metadata: X-Cache, X-Timer, Strict-Transport-Security

Interpretation:

This endpoint provides definitive proof of which Mastodon account is currently authenticated on the device. It includes the unique user ID, username, account creation date, profile metadata, and account attributes. This is a key forensic artifact linking the emulator, session token, and intercepted traffic to a specific Mastodon user.

The screenshot shows the Burp Suite interface with the following details:

Network Tab (HTTP history):

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TLS	IP	Cookies	Time
80	https://mastodon.social	GET	/api/v1/accounts/verify_credentials			200	1990	JSON			✓	151.101.193.91		00:31:	
81	https://mastodon.social	GET	/api/v1/accounts/verify_credentials			200	1990	JSON			✓	151.101.193.91		00:31:	
2920	https://mastodon.social	POST	/api/v1/apps		✓	200	1410	JSON			✓	151.101.65.91		01:19:	
2921	https://mastodon.social	GET	/api/v2/instance			200	25122	JSON			✓	151.101.65.91		01:19:	

Request Panel:

```
Pretty Raw Hex
1 GET /api/v1/accounts/verify_credentials HTTP/2
2 Host: mastodon.social
3 Authorization: Bearer E7JuFR_pj1qlu1CbWV6Bb8KOamH_d-q2hWFYsrJiNeY
4 User-Agent: Tushy/25.0 Android/12 OkHttp/4.12.0
5 Accept-Encoding: gzip, deflate, br
6
7
```

Response Panel:

```
Pretty Raw Hex Render
20 X-Cache-Hits: 0, 0, 0
21 X-Timer: S17638001282.826655,V80,VE280
22 Vary: Authorization, Origin, Accept-Encoding
23 Strict-Transport-Security: max-age=31557600
24 Alt-Svc: h3=":443",ma=86400,h3-27=:443;ma=86400
25
26 {
  "id": "115592369761209551",
  "username": "test605501",
  "acct": "test605501",
  "display_name": "",
  "locked": false,
  "bot": false,
  "discoverable": true,
  "indexable": true,
  "group": false,
  "created_at": "2025-11-22T00:00:00.000Z",
  "note": "",
  "url": "https://mastodon.social/@test605501",
  "uri": "https://mastodon.social/api/users/115592369761209551",
  "avatar": "https://mastodon.social/avatar/original/missing.png",
  "avatar_static": "https://mastodon.social/avatar/original/missing.png",
  "header": "https://mastodon.social/header/original/missing.png",
  "header_static": "https://mastodon.social/header/original/missing.png",
  "followers_count": 0,
  "following_count": 0,
  "statuses_count": 0,
  "last_status_at": null,
  "hide_collections": null,
```

Bottom Status Bar:

Memory: 333.9MB Disabled 1:26 AM ENG US 2025-11-24

Structured JSON Representation

The extracted data was converted into a simplified **forensic JSON record**, highlighting only the critical fields necessary for analysis.

The screenshot shows a dark-themed instance of Visual Studio Code. The left sidebar contains various icons for file operations like Open, Save, Find, and Settings. The top menu bar includes File, Edit, Selection, View, and a separator. A search bar is located at the top center. On the right side, there are icons for minimizing, maximizing, and closing the window.

The main editor area displays a JSON file named "tel.js". The file path is shown as "C:\Users\amrai\OneDrive\Desktop\tel.js". The code itself is a single object with the following properties:

```
1  {
2      "id": "1153596761298551",
3      "username": "test685501",
4      "acct": "test685501",
5      "display_name": "",
6      "discoverable": true,
7      "bot": false,
8      "group": false,
9      "locked": false,
10     "created_at": "2025-11-02T00:00:00.000Z",
11     "avatar": "https://mastodon.social/avatars/original/missing.png",
12     "header": "https://mastodon.social/headers/original/missing.png"
13 }
14
```

The status bar at the bottom indicates the file is 14 lines long, has 4 spaces per tab, and is in UTF-8 encoding. It also shows the current language is JSON and the date and time as 1:58 AM on November 24, 2025.

Example 3: Tusky Login Credentials POST

Endpoint: /auth/sign_in (Mastodon login POST)

Feature Shown: Burp Decoder – Revealing Hidden Login Fields

Host: mastodon.social

Key forensic artifacts extracted: user[email],user[password]

Interpretation:

Tusky sends login data in **URL-encoded form**, which makes the parameters appear unreadable in the raw request. The encoded POST body was decoded in Burp Suite using **Smart Decode**, revealing the hidden login fields.

Results and Insights

Findings:

Through Frida instrumentation and Burp Suite interception, I was able to successfully capture and analyze live HTTPS traffic from both Instagram and Tusky (for Mastodon). The analysis revealed several important forensic artifacts:

- **Hidden internal API endpoints** used for login, session validation, account-family checks, and app registration.
- **Device and session identifiers such as X-Ig-Device-Id and X-Ig-Family-Device-Id**
- **Navigation metadata** such as X-Ig-Nav-Chain, which ties UI events to backend events.
- **Authenticated account information**, including Instagram pk/id, usernames, and Mastodon account fields such as acct, display_name, and account creation timestamps.
- **Login credentials from Tusky**, revealed after decoding URL-encoded POST data using Burp's *Smart Decode*.

These findings confirm that runtime network interception allows an examiner to observe sensitive, normally encrypted data exchanged between Android apps and their backend servers.

How the Exploration Was Useful

This exploration demonstrated a complete, repeatable workflow for forensic network analysis of mobile apps protected by TLS and SSL pinning. The process was useful because it:

- Bypassed SSL pinning, enabling full inspection of encrypted traffic.
- Allowed the extraction of forensic evidence directly linked to user actions, such as login attempts, session creation, and account verification.
- Showed how device identifiers and timestamps can be used to build a forensic timeline.
- Proved that both mainstream apps (Instagram) and federated apps (Tusky/Mastodon) leak meaningful forensic artifacts even over HTTPS.
- Highlighted that URL-encoded login fields can be decoded and reconstructed to reveal sensitive user inputs.
- Provided a practical demonstration of Frida + Burp Suite cooperation, showing how dynamic instrumentation and MITM interception complement each other.

Overall, the project validated that runtime network forensics is an effective method for discovering hidden API behavior, session metadata, and authenticated account details.

Limits of the Research and Exploration:

Several constraints and challenges were encountered during the project:

- **Frida detection and app crashes:** Instagram repeatedly crashed during early attempts.
- **SSL pinning complexity:** Basic Frida scripts were not sufficient; full interception required system-level CA installation using RootAVD + Magisk.
- **Emulator instability:** Different Android images (31, 35, 36) behaved inconsistently, causing hangs, UI freezes, or broken TLS sessions.
- **Technical setup requirements:** The process required multiple tools (Frida-server, Burp Suite, RootAVD, Magisk) and detailed configuration.
- **Use of authorized test accounts:** Only test accounts could be used.
- **Limited scope:** The research focused only on Instagram and Tusky; other apps may use stronger protections or entirely different security models. For example, Telegram relies on **MTProto** rather than standard HTTPS for authentication. As a result, Burp Suite interception and Frida injection were ineffective, and the app failed to log in when the proxy or hooks were active.

Conclusion:

This project successfully demonstrated that runtime network interception, when combined with Frida-based SSL-pinning bypass and Burp Suite MITM analysis, can reveal sensitive forensic artifacts from Android applications even when they communicate over encrypted HTTPS channels. Through controlled testing on Instagram and Tusky, the workflow exposed device identifiers, session tokens, navigation metadata, OAuth credentials, and authenticated account details. These findings confirm that dynamic instrumentation is an effective and reliable method for extracting forensic evidence from real-world mobile applications.

Future work:

- Since Telegram could not be intercepted during this project, a future goal is to investigate the exact cause and attempt a successful MITM capture.
- Test the **Objection** tool to automate SSL-pinning bypass and runtime analysis.
- Explore **Java function hooking** with Frida to capture deeper runtime data inside the apps.

Other Scenarios Where These Insights Apply:

The techniques used in this project can be applied to numerous real-world forensic and security scenarios:

- **Incident response:** verifying whether a compromised device performed certain actions by reconstructing network timelines.
- **App behavior auditing:** understanding hidden or undocumented API endpoints in mobile apps.
- **Digital evidence validation:** confirming login events, device identifiers, and account ownership.

- **Security research and penetration testing:** identifying insecure data handling practices, excessive data leakage, or weaknesses in TLS/SSL implementations.

Broad Conclusions:

This project highlights the broader importance of understanding how mobile applications handle encrypted communication at runtime. The results show that many Android apps rely heavily on backend APIs and exchange significant metadata that becomes observable once proper instrumentation is applied. Dynamic analysis provides a more accurate picture of actual app behavior than static inspection alone, revealing hidden communication patterns, internal endpoints, and sensitive metadata flows. More generally, this work demonstrates the value of combining multiple tools, controlled environments, and systematic testing to produce reliable forensic evidence and to better understand the security, behavior, and data exposure of modern mobile applications.

AI Use Section:

AI Tool Name	Version, Account Type	Specific feature for which the AI tool was used
ChatGPT (GPT-5 Thinking mini)	Plus	<ul style="list-style-type: none"> - Learning about Frida and SSL pinning Bypass in detail. - Learning tool functions and new concepts. - Learning the SSL-pinning JavaScript code. - Understanding how to read and interpret HTTPS request and response details.

Value Addition:

- I read additional resources to better understand the purpose and internal workings of Frida.
- I defined Frida in my own words based on my understanding from ChatGPT and other resources.
- 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.

Appendix:

Explain what is Frida, and what are its uses?

ChatGPT said:

Frida is an open-source dynamic instrumentation toolkit. It lets you inject JavaScript into running processes (Android, iOS, Windows, macOS, Linux) so you can observe and modify what an app does at runtime—without rebuilding it.

What you can do with Frida

- Runtime analysis & debugging: Hook functions/methods, inspect arguments/returns, trace call stacks.
- Security testing: Probe protected code paths, test defenses (e.g., SSL-pinning bypass, anti-tamper/anti-debug), fuzz inputs.
- Reverse engineering: Explore obfuscated logic, discover hidden/undocumented APIs, extract in-memory strings/keys.
- Network forensics: Intercept or log HTTP/TLS calls to capture tokens, headers, URLs in real time (in authorized test setups).
- Malware analysis: Watch file, network, crypto, and persistence behaviors as they happen.
- Automation: Script repeatable hooks to validate hypotheses and generate reports/artifacts.

What is the meaning of reverse engineering

ChatGPT said:

Reverse engineering is the process of analyzing a finished system (software, hardware, or a protocol) to understand how it works internally—its structure, logic, data flows, and interfaces—usually when source code or design docs aren't available.

What it's used for

- Security: find vulnerabilities, verify patches, analyze malware.
- Interoperability: document file formats/APIs to build compatible tools.
- Forensics: reconstruct actions, timelines, and data handling in apps.
- Debugging/maintenance: understand legacy or third-party components.

Common approaches (software)

- Static analysis: study binaries/APKs without running them (e.g., Ghidra/IDA, jadx, apktool).
- Dynamic analysis: observe behavior while running (e.g., Frida hooks, debuggers, sandboxing).
- Hybrid: combine both to validate hypotheses.

Always ensure you have legal authorization—reverse engineering can be restricted by law, licenses, or terms of service.

SSL-pinning bypass?

ChatGPT said:

SSL-pinning bypass is the act of defeating an app's client-side certificate/public-key pinning so the app will accept a different TLS certificate (for example, a proxy's certificate). This allows you to intercept and inspect the app's encrypted network traffic (headers, tokens, URLs) during testing or forensics — but only on apps/devices you own or are authorized to test.

Quick breakdown

- **SSL pinning:** app verifies the server's certificate or public key against a known value baked into the app. If it doesn't match, the connection is rejected.
- **Bypass:** modify or neutralize the pinning check so the app treats a proxy's certificate as valid and continues the TLS handshake.

Why you might do it (legitimate reasons)

- Debugging and troubleshooting network calls.
- Security testing / pentesting to validate server-side protections.
- Forensic collection of runtime artifacts in a controlled lab or with consent.

Common bypass approaches (high-level)

- **Hooking/patching the app's pin-check functions** (e.g., okhttp3.CertificatePinner.check, X509TrustManager.checkServerTrusted) and return success.
- **Hooking platform APIs** (e.g., SecTrustEvaluate on iOS) to force trust.
- **Installing a custom root CA** on the device (when the app doesn't pin).
- **Using dynamic instrumentation** tools like Frida to inject code at runtime and disable or override checks.
- **Patching the binary** (static modification) to remove or alter pinning logic.

Risks & cautions

- May expose sensitive user data — treat captured artifacts as highly sensitive.
 - Illegal/unethical without permission; may violate ToS or laws.
 - Can break app functionality if done incorrectly.
-

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 (Conscript, 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.

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:akabe1/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:akabe1/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.

universal SSL pinning bypass for Android apps.

Goal:

② **Force the app to trust your Burp certificate**, so you can see HTTPS traffic in Burp even if the app uses certificate pinning.

What the code does step by step (in simple words)

Runs inside the app's Java world

```

setTimeout(function(){
    Java.perform(function (){


```

```
...  
});  
, 0);
```

Java.perform = “run this code in the app’s Java VM”

setTimeout(..., 0) = just schedules it to run when the app is ready

Loads your Burp CA certificate from the device

```
var fileInputStream = FileInputStream.$new("/data/local/tmp/cert-der.crt");var bufferedInputStream  
= BufferedInputStream.$new(fileInputStream);var ca = cf.generateCertificate(bufferedInputStream);
```

It reads /data/local/tmp/cert-der.crt (your Burp cert in DER format)

Turns it into a Java X509Certificate object

This is the **CA we want the app to trust**

Creates a fake KeyStore that trusts your CA

```
var keyStore = KeyStore.getInstance(keyStoreType);  
keyStore.load(null, null);  
keyStore.setCertificateEntry("ca", ca);
```

Makes an in-memory KeyStore

Puts your CA inside it

This is like saying: “Here is a trust store that trusts only Burp’s cert”

Builds a TrustManager that uses that KeyStore

```
var tmf = TrustManagerFactory.getInstance(tmfAlgorithm);  
tmf.init(keyStore);  
  
tmf = TrustManagerFactory
```

It creates TrustManagers that **trust your CA** instead of the app’s original pinned cert

Hooks the Java SSLContext.init() function

This is the **Java hook** part:

```
SSLContext.init.overload(  
    "[Ljavax.net.ssl.KeyManager;",  
    "[Ljavax.net.ssl.TrustManager;",  
    "java.security.SecureRandom"  
)  
.implementation = function(a, b, c) {  
    console.log("[o] App invoked javax.net.ssl.SSLContext.init...");  
    SSLContext.init.overload(...).call(this, a, tmf.getTrustManagers(), c);  
    console.log("[+] SSLContext initialized with our custom TrustManager!");  
}
```

Normally the app calls `SSLContext.init(keyManagers, trustManagers, random)` with its own **trust managers** (pinned certs).

This hook **intercepts that call and replaces** `trustManagers` **with** `tmf.getTrustManagers()`, which use *your KeyStore*.

Result: the app now **trusts your Burp CA** → SSL pinning is bypassed.

Yes: **this is exactly “hooking a Java function with Frida.”** ✓

[How you use it \(in practice\)](#)

Put Burp CA on device and convert to DER (cert-der.crt)

Push it to /data/local/tmp/cert-der.crt

Run app with Frida:

`frida -U -f com.instagram.android -l frida-android-repinning.js --no-pause`

Then check Burp → you should see decrypted HTTPS.

OpenSSL

ChatGPT said:

You probably mean **OpenSSL** ☐ (people typo it as “OpenSLL” all the time).

OpenSSL is:

A **software library** (mostly written in C)

Used to implement **encryption on the internet**: TLS/SSL for HTTPS, VPNs, mail servers, etc.

Installed on most Linux/Unix systems and used by tons of programs (web servers like Apache/Nginx, curl, Python, etc.).

It mainly provides:

Crypto algorithms

Symmetric: AES, ChaCha20, etc.

Asymmetric: RSA, ECDSA, etc.

Hashes: SHA-256, SHA-1, etc.

TLS/SSL protocols

Handles the handshake, certificates, key exchange → basically what happens when you see https://.

Command-line tool openssl

Super handy for:

Generating keys:

```
openssl genrsa -out key.pem 2048
```

Creating CSRs and self-signed certs

Converting cert formats (PEM, DER, PFX)

Testing servers:

```
openssl s_client -connect example.com:443
```

Encrypting/decrypting small files, computing hashes, etc.

So in your SSL-pinning / Burp / Android world, OpenSSL is one of the low-level libraries that can be used by apps or systems to **do the actual TLS crypto**.

If you want, next I can show you a few openssl commands that are useful for hacking/CTF/mobile-testing work.

You said:

OpenSSL for Burp.cer

ChatGPT said:

If you're asking how **OpenSSL is used with** burp.cer, it usually means you want to work with **Burp Suite's CA certificate**.

This certificate is what you install on your Android device so Burp can decrypt HTTPS traffic.

Let me break it down clearly and show what OpenSSL is used for.

✓ What is burp.cer?

burp.cer is the **Burp CA certificate** generated by Burp Suite.

You install it on your device so that:

Burp becomes a trusted Certificate Authority

Your Android device will accept Burp's fake certificates

This allows Burp to intercept **HTTPS traffic (MITM)**

If the certificate is not trusted → your device/app will show SSL errors → HTTPS interception won't work.

✓ What is OpenSSL used for?

OpenSSL is used to **convert, inspect, or extract** information from burp.cer.

For example:

① 1. Convert Burp certificate to PEM

Burp sometimes exports the certificate in DER format (.cer).

You may need PEM (.pem) for tools like adb, Android system trust store, Frida scripts, etc.

Convert using OpenSSL:

```
openssl x509 -inform DER -in burp.cer -out burp.pem
```

② 2. View certificate info

```
openssl x509 -in burp.cer -inform DER -text -noout
```

This lets you see:

Subject

Issuer

SHA-256 fingerprint

Validity dates

Public key

Signature algorithm

¶ 3. Convert to a system-trust store format (Android older versions)

Android sometimes needs **.crt**:

```
openssl x509 -in burp.cer -inform DER -out burp.crt
```

¶ 4. Generate Frida's trust patching certificates (if doing SSL unpinning)

Some Frida SSL bypass scripts require the certificate in **PEM format**.

Again:

```
openssl x509 -inform DER -in burp.cer -out burp.pem
```

Then the script can inject the PEM at runtime.

□ IMPORTANT FOR ANDROID 7+ (Nougat and above)

Android does **NOT trust user-installed certificates for apps by default**.
Apps must explicitly allow **user-added CAs** in their Network Security Config.

If they don't → even if you install burp.cer, the MITM won't work.

That's why tools like **Frida SSL Pinning bypass** or **MagiskTrustUserCerts** are required.

★ Summary

Goal

OpenSSL Command

Goal	OpenSSL Command
Inspect the certificate	openssl x509 -in burp.cer -inform DER -text -noout
Convert .cer → .pem	openssl x509 -inform DER -in burp.cer -out burp.pem
Convert .cer → .crt	openssl x509 -inform DER -in burp.cer -out burp.crt

If you tell me what exactly you want to do with burp.cer, I can give the exact commands:

Are you trying to **install Burp CA on Android?**

Convert it for **Frida SSL bypass?**

Use it inside **Magisk trust store?**

Patch APK network config?

Just tell me your goal — I'll give step-by-step instructions.

You said:

trying to install Burp CA on Android as system cer

ChatGPT said:

Perfect — installing **Burp CA as a system certificate** is the correct approach for Android 7+ when you want all apps (even those that ignore user certificates) to trust Burp.

Below is the **exact step-by-step method** depending on whether your Android is **rooted (Magisk)** or **unrooted**.

✓ If your Android Emulator / phone is ROOTED (Magisk / AVD + root images)

This is the clean and correct way.

1. Export Burp certificate as DER

In Burp:

Proxy → Proxy Settings → CA Certificate → Export → DER format

Save as:

burp.cer

2. Convert it to PEM (required for system store)

Android system certs must be:

PEM format

Named with the certificate's subject hash + .0

Run:

```
openssl x509 -inform DER -in burp.cer -out burp.pem
```

3. Compute Android's certificate hash

```
openssl x509 -inform PEM -subject_hash_old -in burp.pem | head -1
```

Example output:

```
9a5ba575
```

That means the file must be named:

```
9a5ba575.0
```

Rename:

```
mv burp.pem 9a5ba575.0
```

4. Push it to Android's system CA directory

Newer Android:

```
/system/etc/security/cacerts/
```

Older Android:

```
/system/etc/security/cacerts/
```

Push the certificate:

```
adb root
```

```
adb remount
```

```
adb push 9a5ba575.0 /system/etc/security/cacerts/
```

5. Fix file permissions (very important)

Android requires:

owner: rootgroup: rootpermissions: 644

Run:

```
adb shell chmod 644 /system/etc/security/cacerts/9a5ba575.0
```

```
adb shell chown root:root /system/etc/security/cacerts/9a5ba575.0
```

6. Reboot

adb reboot

After reboot, Burp is now a *system-trusted CA*.

All apps (including those with network security config) will trust it unless they implement **certificate pinning**.

Testing it worked

On the device:

Open Chrome → https://burp

If it loads Burp's certificate → SUCCESS

Then try intercepting HTTPS again.

If Your Android Device Is NOT Rooted

Android **WILL NOT** trust a user-installed certificate for most apps.
Only the system CA store is fully trusted.

Your options:

1. Use Magisk + MagiskTrustUserCerts module

This automatically copies your user CA to the system CA store.

2. Or use Frida SSLPinning bypass instead

For apps that enforce pinning.

Note about Android Emulator

The **standard AVD emulator is not rooted** unless you:

enable root in emulator settings (API 28+),

or use Google's **rooted images**,

or use **Genymotion rooted**.

To check:

adb root

If it says:

adb cannot run as root in production builds

→ emulator is **not** rooted.

If you want, I can walk you through installing Burp CA *specifically on your emulator model (Android_31_x86_64, etc.)* step-by-step.

Just tell me:

Your emulator Android version?

Does adb root work?

You said:

adb root work

ChatGPT said:

Great — since adb root **works**, your emulator is rooted and you can install Burp's CA as a **system certificate** □

Below are the exact steps you should follow. This will make *all* apps trust Burp (except those with certificate pinning).

Step-by-Step: Install Burp CA as a System Certificate (Rooted Emulator)

1□ Export Burp CA certificate from Burp

In Burp:

Proxy → Proxy Settings → CA Certificate → Export → Certificate in DER format

Save it as:

burp.cer

2 □ Convert .cer → .pem (required for Android system store)

Open a terminal in the directory where your cert is and run:

```
openssl x509 -inform DER -in burp.cer -out burp.pem
```

3 □ Get the certificate hash (VERY important)

```
openssl x509 -inform PEM -subject_hash_old -in burp.pem | head -1
```

Example output:

9a5ba575

Whatever number appears → that is the required system filename.

Rename:

```
mv burp.pem 9a5ba575.0
```

¶ **The .0 extension is mandatory.**

4 □ Push the certificate into the Android system CA store

System CA directory (modern Android: API 26+):

/system/etc/security/cacerts/

Run:

```
adb root
```

```
adb remount
```

```
adb push 9a5ba575.0 /system/etc/security/cacerts/
```

If successful, you should see:

```
9a5ba575.0: 1 file pushed
```

5□ Fix ownership + permissions

Android requires CA files to be:

owned by **root:root**

permission **644**

Run:

```
adb shell chown root:root /system/etc/security/cacerts/9a5ba575.0
```

```
adb shell chmod 644 /system/etc/security/cacerts/9a5ba575.0
```

6□ Reboot the emulator

```
adb reboot
```

7□ Test the installation

On the emulator, open Chrome and visit:

<https://burp>

If it loads the Burp Suite Certificate page **without any SSL error**, your installation is perfect.

Now all apps that **do not use certificate pinning** will trust Burp automatically.

OPTIONAL: Disable certificate pinning (if needed)

Some apps still reject MITM even with system CA because they use **SSL pinning**.
In those cases, you need:

Frida SSL pinning bypass

Or patching the APK (Objection / Apktool)

Or bypass modules like **JustTrustMe** (Magisk)

But installing the Burp CA as system cert is the required first step.

Instagram Headers – What they mean (simple)

X-Ig-Device-Id

A unique ID Instagram uses for this device.

□ Forensics: links activity to *this* device.

X-Ig-Family-Device-Id

Another long-term ID used across Meta “family” apps (IG, FB, etc.).

- Forensics: can link multiple apps/accounts to the same device.

X-Ig-Android-Id

ID based on the Android system/device.

- Forensics: supports attribution to a specific Android install.

X-Pigeon-Session-Id

Session identifier for this app session.

- Forensics: groups multiple requests into one “login/use session”.

X-Pigeon-Rawclienttime

Client-side timestamp (from the phone) when the request was made.

- Forensics: helps build a **timeline** of user actions.

X-Ig-Nav-Chain

“Navigation chain” – shows which screens / flows the user went through.

- Forensics: reconstructs what the user clicked or opened.

X-Fb-Connection-Type / X-Ig-Connection-Type

How the device is connected – e.g., WIFI, MOBILE.

- Forensics: tells you network type used during the action.

X-Ig-Origin-Region

Region / datacenter handling the request (e.g., country/region code).

- Forensics: rough geographic / infrastructure info.

X-Fb-Client-Ip-Forwarded

The client IP as seen by their edge/server.

- Forensics: which IP address the request really came from.

X-Fb-Server-Cluster-Forwarded

Which backend server/cluster processed the request.

- Mostly infra info, but can support log correlation.

Tusky + OAuth – in simple words

Tusky is just a **Mastodon client**, and it uses **OAuth** to log in.

1. App registration (</api/v1/apps>)

Tusky sends:

client_name → Tusky

redirect_uris → where Mastodon sends the user back

scopes → what permissions it wants (read, write, follow, etc.)

Server returns:

client_id

client_secret

vapid_key

Forensics: these are the **app credentials** Tusky uses to talk to the server.

2. Login / token

Later, Tusky sends **username + password** (your Example 3) → server returns an **access token**.

Authorization: Bearer <token> in later requests

Forensics: this token proves which account is authenticated and lets you tie *all* later Tusky requests to that user.

References:

[How to Detect Frida Toolkit Abuse in Your Mobile App](#)

[Welcome | Frida • A world-class dynamic instrumentation toolkit](#)

[SSL Pinning Bypass: Essential Methods and Security Solutions](#)

[Step-by-Step Guide to Bypass SSL Pinning with Frida - DEV Community](#)

[okankurtuluss/FridaBypassKit: Android Universal Anti-Bypass Framework - Root, SSL Pinning, Emulator & Debug Detection Bypass](#)

[Releases · frida/frida](#)

[Cracking the Code: Mastering SSL Pinning Bypass with Frida for Pentesting Android Apps | by Ravneet Singh | Medium](#)

[LevelUp 0x04 - Fun with Frida on Mobile](#)

[Android SSL Pinning Bypass for Bug Bounties & Penetration Testing](#)

[Bypassing Android SSL Pinning with FRIDA » SecurityGrind](#)

[Hail Frida!! The Universal SSL pinning bypass for Android applications | by Vedant | InfoSec Write-ups](#)

[How Does Frida Work In Mobile Application Security Testing? - Cyberly](#)

[newbit / rootAVD · GitLab](#)

[Releases · NVISOsecurity/AlwaysTrustUserCerts](#)

[Bypass SSL Pinning Using Frida](#)

[Mirror frida-android-repinning.js](#)

[Android SSL Pinning Bypass](#)

[Burp Suite and Frida on an Android Emulator](#)

[Installing \(AND ROOTING\) Android Emulator \[2024 UPDATE\]](#)

[Android SSL Pinning Bypass | Magisk, Frida & Burp Suite Configuration for Android Studio | BrutDroid - YouTube](#)

[Using Burp Suite with an Android emulator](#)

[Disable SSL Pinning for any Android application New method for split APK's](#)

Bypass SSL Pinning on a Real Device — Burp Suite + Magisk + Frida (Step-by-Step)

Defeat SSL Pinning | Android Security Crash Course

Android SSL Pinning Bypass for Bug Bounties & Penetration Testing

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.
06-11-2025	6	Captured, organized, and analyzed intercepted request and response traffic; extracted the first example of forensic data and studied JSON.
07-11-2025	6	Analyzed additional captured data; documented the second and third example outputs; began writing detailed results and the JSON structures.
08-11-2025	3	Continued refining the analysis and writing of the report for the three examples.
09-11-2025	4	Attempted intercepting Telegram; the attempt was unsuccessful.
13-11-2025	2	Worked on completing the Progress Report Template.
20-11-2025	3	Continued attempting Telegram interception. Frida successfully injected, but Burp Suite did not capture any traffic. Multiple attempts were unsuccessful.
21-11-2025	5	Tested interception on several other apps: Twitter, Facebook, Facebook Lite, Messenger, Signal, Vero, Reddit, and Keep. All either refused to log in, failed to connect, or operated normally without producing any traffic in Burp Suite. SSL-pinning bypass was successful only in two apps: Tusky and Pinterest.
21-11-2025	3	Captured, organized, and analyzed intercepted request and response traffic for Tusky; extracted two forensic examples and examined the JSON output.
22-11-2025	3	Analyzed additional captured data; documented the third example output; began working on the final report.
23-11-2025	4	Continued working on the final report.
24-11-2025	3	Finalized the final report.
25-11-2025	6	Collected the information needed to prepare the video and practiced the demonstration steps.
28-11-2025	4	Practiced the video demo, recorded it, and finalized it.

Total Hours : 107

Student 2: Team member

Title : Voice and Audio Artifact Extractor

Student Name: Shiksha Kaharanthilake

Student ID: 300398278

Abstract:

This project explores the extraction and analysis of voice-based artifacts from **WhatsApp and Messenger** communications. Using **Whisper** for transcription and **Pydub/FFmpeg** for audio conversion, the system automates preprocessing, noise reduction, and batch transcription to generate structured outputs. The workflow enhances forensic examination by converting raw voice notes into searchable, analyzable digital artifacts.

Background Research

Related Tools or Research

Audio forensics and communication-artifact extraction have become essential in digital investigations. Modern communication apps such as WhatsApp and Messenger increasingly rely on voice messaging, requiring tools that can reliably convert and analyze spoken content. Existing research emphasizes:

- **Speech-to-text using deep learning models** (Whisper, Wav2Vec2)
- **Signal processing methods** for noise reduction and audio normalization
- **Automated forensic reporting systems** for large-scale communication analysis

Whisper, developed by OpenAI, is widely recognized for its robustness in challenging audio conditions such as low volume, background noise, and compressed formats (e.g: *.opus*). Combined with Pydub and FFmpeg, Whisper enables accurate transcription from heterogeneous audio sources.

Purpose of the Tool / Forensic Examination

The purpose of the *Voice and Audio Artifact Extractor* is to enable:

- **Automated extraction of text-based artifacts** from voice notes

- **Conversion of raw audio from multiple apps** into normalized formats
- **Batch processing** for large datasets
- **Error-handled transcription** suitable for forensic review
- **Structured outputs** usable in reporting, visualization, or further investigation

The system reduces manual effort and improves accuracy when analyzing voice messages from digital communications.

Forensic Scenarios Where the Tool Is Useful

Scenario: Multi-Platform Communication Investigation

Imagine a case involving an individual who communicates through voice messages on both WhatsApp and Messenger. Investigators must retrieve, convert, and analyze hundreds of voice notes. Manual handling is slow and error-prone.

This tool assists by:

- Automatically converting *.opus*, *.m4a*, *.mp3*, and other formats
- Normalizing and reducing noise
- Performing uniform transcription
- Producing text files and JSON evidence summaries

This supports investigations involving cyber harassment, fraud, coordinated planning, or unstructured voice-based communication.

Forensic Plan

How the Tools Work

Whisper

- Deep-learning speech-recognition system
- Handles multiple languages, accents, and noisy environments
- Produces transcripts with high accuracy

Pydub

- Python library for manipulating audio
- Enables conversion between formats (e.g., OPUS → WAV)
- Supports trimming, normalization, and segmentation

FFmpeg

- Backend engine for decoding/encoding
- Required by Pydub for full codec support

Together, these tools form a pipeline that automates extraction and transcription

Examination Methodology

1. Collection of Audio Artifacts

- Export WhatsApp voice notes
- Collect Messenger audio files
- Organize in structured folders

2. Preprocessing

- Convert all files into WAV (Whisper-compatible)
- Normalize volume
- Reduce background noise

3. Error Handling

- Skip or log corrupted files
- Detect missing or zero-byte recordings

4. Whisper Transcription

- Test multiple models (tiny, base, small)
- Select the most accurate model (“base” chosen)

5. Batch Processing

- Process all audio files sequentially
- Produce transcripts with consistent formatting

6. Output Generation

- Save transcripts as .txt and .json files

Produce structured evidence summaries

Detailed Analysis

3.1 Environment Setup

Steps completed:

- Installed Whisper, Pydub, FFmpeg
- Configured PATH variables
- Installed Whatsapp and Messenger using a compatible .apk

```
C:\Users\niman>whisper -help
usage: whisper [-h] [--model MODEL] [--model_dir MODEL_DIR] [--device DEVICE] [--output_dir OUTPUT_DIR]
               [--output_format {txt,vtt,srt,tsv,json,all}] [--verbose VERBOSE] [--task {transcribe,translate}]
               [--language {af,am,ar,as,az,ba,be,bg,bn,bo,bs,ca,cs,cy,da,de,el,en,es,et,eu,fa,fi,fo,fr,gl,g
               ,hy,id,is,it,ja,jw,ka,kk,km,ko,la,lb,ln,lo,lt,lv,mg,mi,mk,mn,mr,ms,mt,my,ne,nl,no,oc,pa,pl,ps,pt,ro,r
               ,sq,sr,su,sv,sw,ta,te,tg,th,tk,tl,tr,tt,uk,ur,uz,vi,yi,yo,yue,zh,Afrikaans,Albanian,Amharic,Arabic,Armenian,Ass
               ir,Basque,Belarusian,Bengali,Bosnian,Breton,Bulgarian,Burmese,Cantonese,Castilian,Catalan,Chinese,Croatian,Cze
               ,Estonian,Faroese,Finnish,Flemish,French,Galician,Georgian,German,Greek,Gujarati,Haitian,Haitian Creole,Hausa,
               ungarian,Icelandic,Indonesian,Italian,Japanese,Javanese,Kannada,Kazakh,Khmer,Korean,Lao,Latin,Latvian,Letzebur
               n,Luxembourgish,Macedonian,Malagasy,Malay,Malayalam,Maltese,Mandarin,Maori,Marathi,Moldavian,Moldovan,Mongolia
               n,Nynorsk,Occitan,Punjabi,Pashto,Persian,Polish,Portuguese,Punjabi,Pushto,Romanian,Russian,Sanskrit,Serbian,
               nhalese,Slovak,Slovenian,Somali,Spanish,Sundanese,Swahili,Swedish,Tagalog,Tajik,Tamil,Tatar,Telugu,Thai,Tibeta
               n,Urdu,Uzbek,Vlencian,Vietnamese,Welsh,Yiddish,Yoruba}]
               [--temperature TEMPERATURE] [--best_of BEST_OF] [--beam_size BEAM_SIZE] [--patience PATIENCE]
```

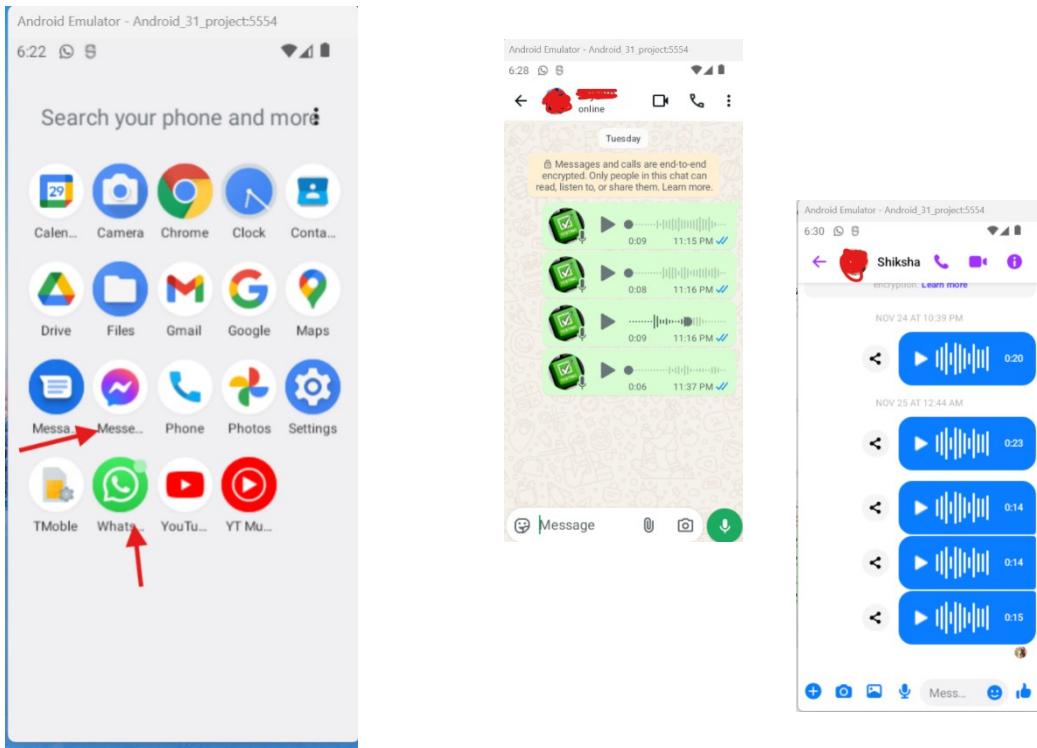
Whisper

```
Command Prompt
C:\Users\niman>pip list | findstr pydub
pydub          0.25.1
```

pydub installed

```
Command Prompt
C:\Users\niman>ffmpeg -version
ffmpeg version 8.0-essentials_build-www.gyan.dev Copyright (c) 2000-2025 the FFmpeg developers
built with gcc 15.2.0 (Rev8, Built by MSYS2 project)
configuration: --enable-gpl --enable-version3 --enable-static --disable-autodetect --enable-fontconfig --enable-
iconv --enable-gnutls --enable-libxml2 --enable-gmp --enable-bzlib --enable-tzma --enable-zlib --enable-libsr
t --enable-libssh --enable-libzmq --enable-avisynth --enable-sdl2 --enable-libwebrp --enable-libx264 --enable-
libx265 --enable-libxvid --enable-libaom --enable-libopenjpeg --enable-libvpx --enable-mediafoundation --enable-
libass --enable-libfreetype --enable-libfribidi --enable-libharfbuzz --enable-libvidstab --enable-libvmaf --enable-
libzimg --enable-amf --enable-cuda-llvm --enable-cuvid --enable-d3d11va --enable-d3d12va --enable-fnvcode
c --enable-libvpl --enable-nvdec --enable-nvenc --enable-vaaPI --enable-openal --enable-libgme --enable-
libopenmp --enable-libopencore-amrwb --enable-libmp3lame --enable-libtheora --enable-libvo-amrwbenc --enable-
libgsm --enable-libopus --enable-libspeex --enable-libvorbis --enable-librubberband
libavutil      60. 8.100 / 60. 8.100
libavcodec     62. 11.100 / 62. 11.100
libavformat    62. 3.100 / 62. 3.100
libavdevice    62. 1.100 / 62. 1.100
libavfilter     11. 4.100 / 11. 4.100
libavscale      9. 1.100 / 9. 1.100
libswresample   6. 1.100 / 6. 1.100
Exiting with exit code 0
C:\Users\niman>
```

ffmpeg



3.2 Audio Collection & Organization

- Gathered WhatsApp voice notes in .opus format(pulled voice notes from sdcard, since all voices are stored in sdcard)

Using command-

```
adb pull "/sdcard/Android/media/com.whatsapp/WhatsApp/Media/WhatsApp Voice Notes" "D:\My Project\Audio\WhatsApp"
```

-

- Downloaded Messenger voice clips (often .m4a)(pulled voices from data/data folder after rooting the device)

Using command-

```
adb pull "/data/data/com.facebook.orca/files/audio/v2.ols100.1" "D:\My Project\Audio\Messenger"
```

- Organized directories by platform and date

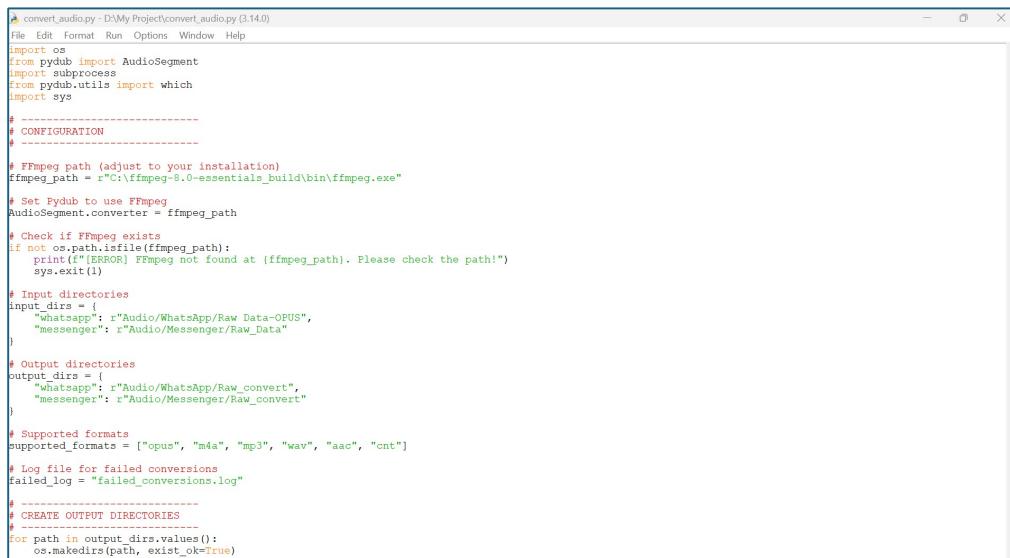
This ensured easy iteration during batch processing.

3.4 Audio Conversion Process

Using Pydub + FFmpeg:

- Loaded audio in its native format
- Exported to WAV at 16-bit PCM (Whisper requirement)
- Normalized volume levels
- Identified problematic or corrupted files

This improved Whisper's accuracy substantially.



```
convert_audio.py - D:\My Project\convert_audio.py (3.14.0)
File Edit Format Run Options Window Help
import os
from pydub import AudioSegment
import subprocess
from pydub.utils import which
import sys

# -----
# CONFIGURATION
#
# FFmpeg path (adjust to your installation)
ffmpeg_path = r"C:\ffmpeg-8.0-essentials_build\bin\ffmpeg.exe"

# Set Pydub to use FFmpeg
AudioSegment.converter = ffmpeg_path

# Check if FFmpeg exists
if not os.path.isfile(ffmpeg_path):
    print("![ERROR] FFmpeg not found at (ffmpeg_path). Please check the path!")
    sys.exit(1)

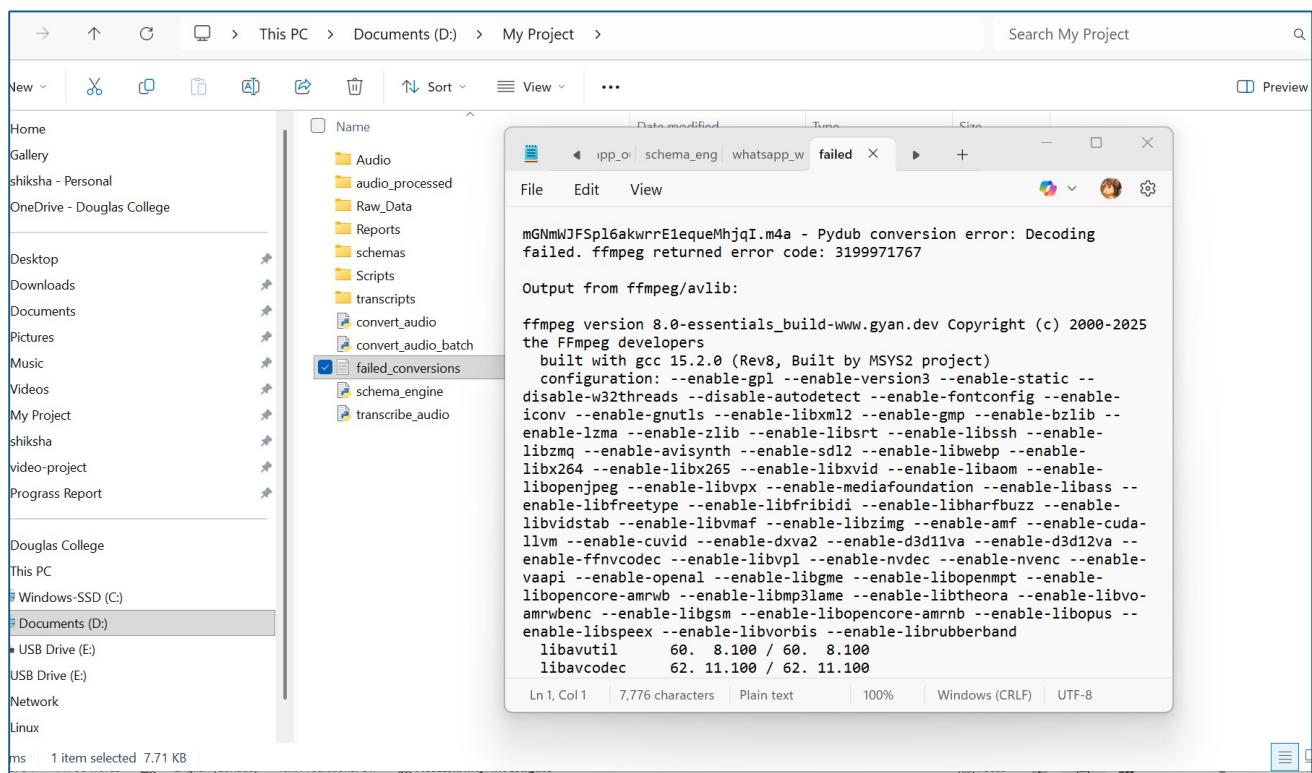
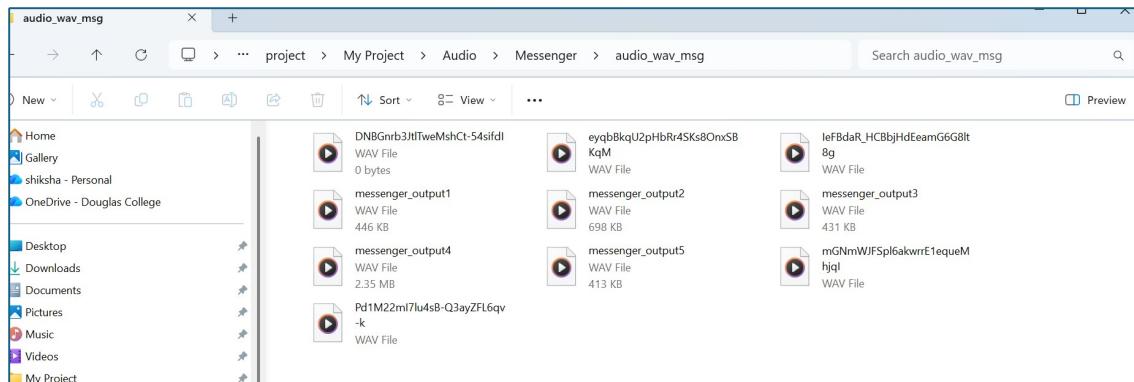
# Input directories
input_dirs = [
    "WhatsApp": r"Audio/WhatsApp/Raw_Data-OPUS",
    "Messenger": r"Audio/Messenger/Raw_Data"
]

# Output directories
output_dirs = [
    "WhatsApp": r"Audio/WhatsApp/Raw_convert",
    "Messenger": r"Audio/Messenger/Raw_convert"
]

# Supported formats
supported_formats = ["opus", "m4a", "mp3", "wav", "aac", "cnt"]

# Log file for failed conversions
failed_log = "failed_conversions.log"

# -----
# CREATE OUTPUT DIRECTORIES
#
for path in output_dirs.values():
    os.makedirs(path, exist_ok=True)
```



3.5 Batch Processing Implementation

Pipeline features:

- Loop through all audio folders
- Try/except blocks to catch conversion or decoding errors
- Logging for unsupported or unreadable files
- Automatic output naming (“msg_01.wav” → “msg_01.txt”)

Batch_processing python script:

```
#!/bin/usr/python

import os
from pydub import AudioSegment
import subprocess
import sys

# -----
# CONFIGURATION
# -----


ffmpeg_path = r"C:\ffmpeg-8.0-essentials_build\bin\ffmpeg.exe"
AudioSegment.converter = ffmpeg_path

if not os.path.isfile(ffmpeg_path):
```

```
print(f"[ERROR] FFmpeg not found at {ffmpeg_path}")
sys.exit(1)

input_root = r"Audio"
output_root = r"audio_processed"
os.makedirs(output_root, exist_ok=True)

supported_formats = ["opus", "m4a", "mp3", "wav", "aac", "cnt"]
failed_log = "failed_conversions.log"

# -----
# HELPER FUNCTIONS
# -----
def log_failure(filename, reason):
    with open(failed_log, "a") as f:
        f.write(f'{filename} - {reason}\n')

def convert_cnt(file_path, output_file):
    for codec in ["aac", "opus"]:
        try:
            subprocess.run(
                [ffmpeg_path, "-y", "-f", codec, "-i", file_path, "-ac", "1", "-ar", "16000",
                 output_file],
                check=True,
                stdout=subprocess.DEVNULL,
                stderr=subprocess.DEVNULL
            )
        except subprocess.CalledProcessError as e:
            log_failure(file_path, str(e))
```

```
        return True

    except subprocess.CalledProcessError:
        continue

    return False


def convert_opus(file_path, output_file):
    try:
        subprocess.run(
            [ffmpeg_path, "-y", "-i", file_path, "-ac", "1", "-ar", "16000", output_file],
            check=True,
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL
        )
        return True
    except subprocess.CalledProcessError:
        return False


# -----
# MAIN BATCH CONVERSION LOOP
# -----


allowed_folders = ["audio_wav_msg", "audio_wav_whatsapp"]

for root, dirs, files in os.walk(input_root):
    if os.path.basename(root) not in allowed_folders:
```

```
        continue
```

```
for filename in files:
```

```
    file_path = os.path.join(root, filename)
    name, ext = os.path.splitext(filename)
    ext = ext[1:].lower()
    output_file = os.path.join(output_root, f'{name}.wav')
```

```
if ext not in supported_formats:
```

```
    print(f'[SKIP] Unsupported: {filename}')
    log_failure(filename, "Unsupported format")
    continue
```

```
if ext == "opus" and os.path.basename(root) == "audio_wav_whatsapp":
```

```
    if convert_opus(file_path, output_file):
        print(f'[OK] Converted {filename}')
    else:
        print(f'[ERROR] Failed {filename}')
        log_failure(filename, "WhatsApp .opus conversion failed")
    continue
```

```
if ext == "cnt" and os.path.basename(root) == "audio_wav_msg":
```

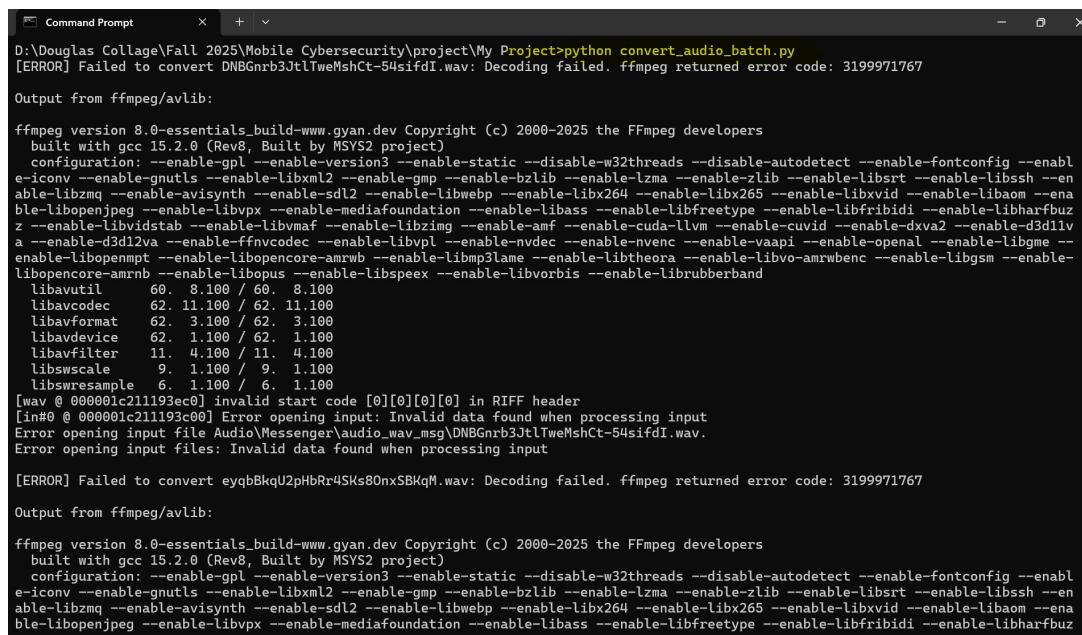
```
    if convert_cnt(file_path, output_file):
        print(f'[OK] Converted {filename}')
    else:
        print(f'[ERROR] Failed {filename}')
        log_failure(filename, "Messenger .cnt conversion failed")
    continue
```

try:

```
    audio = AudioSegment.from_file(file_path, format=ext)
    audio = audio.set_channels(1).set_frame_rate(16000)
    audio.export(output_file, format="wav")
    print(f"[OK] Converted {filename}")
```

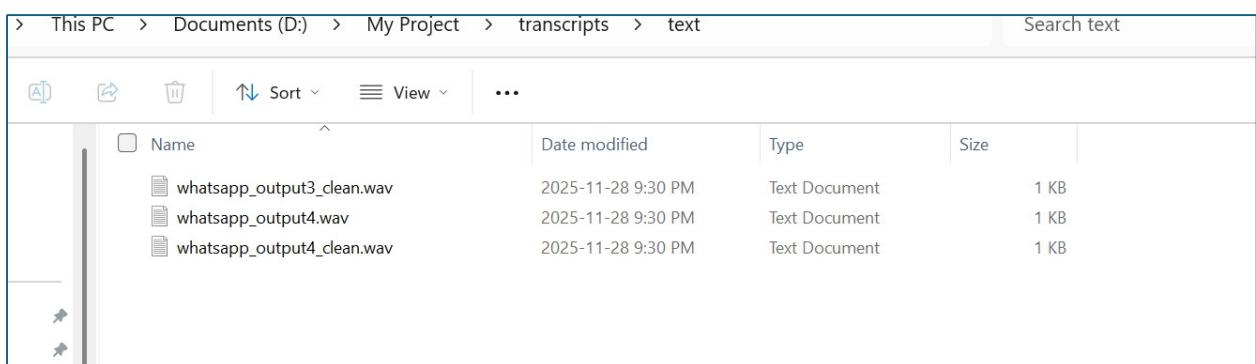
except Exception as e:

```
    print(f"[ERROR] Failed to convert {filename}: {e}")
    log_failure(filename, f"Pydub conversion error: {e}")
```



```
D:\Douglas Collage\Fall 2025\Mobile Cybersecurity\project\My Project>python convert_audio_batch.py
[ERROR] Failed to convert DNBGnrb3JtlTweMshCt-54sifdI.wav: Decoding failed. ffmpeg returned error code: 3199971767
Output from ffmpeg/avlib:
ffmpeg version 8.0-essentials_build-www.gyan.dev Copyright (c) 2000-2025 the FFmpeg developers
  built with gcc 15.2.0 (Rev8, Built by MSYS2 project)
  configuration: --enable-gpl --enable-version3 --enable-static --disable-w32threads --disable-autodetect --enable-fontconfig --enable-iconv --enable-gnutls --enable-libxml2 --enable-gmp --enable-bzlib --enable-lzma --enable-zlib --enable-libsrtp --enable-libssh --enable-libzmq --enable-avisynth --enable-sdl2 --enable-libwebp --enable-libx264 --enable-libx265 --enable-libxvid --enable-libxvidstab --enable-libvmaf --enable-libzimg --enable-amf --enable-cuda-llvm --enable-cuvid --enable-dxva2 --enable-d3d11va --enable-d3d12va --enable-ffnvencodec --enable-libvpxl --enable-nvdec --enable-nvenc --enable-vaaPI --enable-openal --enable-libgme --enable-libopenpnmpt --enable-libopencore-amrnb --enable-libmp3lame --enable-libtheora --enable-libvo-amrwbenc --enable-libgsm --enable-libopus --enable-libspeex --enable-libvorbis --enable-librubberband
  libavutil      60. 8.100 / 60. 8.100
  libavcodec     62. 11.100 / 62. 11.100
  libavformat    62.  3.100 / 62.  3.100
  libavdevice    62.  1.100 / 62.  1.100
  libavfilter     11.  4.100 / 11.  4.100
  libswscale      9.  1.100 /  9.  1.100
  libswresample   6.  1.100 /  6.  1.100
[wav @ 000001c211193ec0] invalid start code [0][0][0][0] in RIFF header
[in@# 0 000001c211193c0] Error opening input: Invalid data found when processing input
Error opening input file Audio\Messenger\audio_wav_msg\DNBGnrb3JtlTweMshCt-54sifdI.wav.
Error opening input files: Invalid data found when processing input
[ERROR] Failed to convert eyqbBkqU2pHbRr4SKs8OnxSBKqM.wav: Decoding failed. ffmpeg returned error code: 3199971767
Output from ffmpeg/avlib:
ffmpeg version 8.0-essentials_build-www.gyan.dev Copyright (c) 2000-2025 the FFmpeg developers
  built with gcc 15.2.0 (Rev8, Built by MSYS2 project)
  configuration: --enable-gpl --enable-version3 --enable-static --disable-w32threads --disable-autodetect --enable-fontconfig --enable-iconv --enable-gnutls --enable-libxml2 --enable-gmp --enable-bzlib --enable-lzma --enable-zlib --enable-libsrtp --enable-libssh --enable-libzmq --enable-avisynth --enable-sdl2 --enable-libwebp --enable-libx264 --enable-libx265 --enable-libxvid --enable-libxvidstab --enable-libvmaf --enable-libzimg --enable-amf --enable-cuda-llvm --enable-cuvid --enable-dxva2 --enable-d3d11va --enable-d3d12va --enable-ffnvencodec --enable-libvpxl --enable-nvdec --enable-nvenc --enable-vaaPI --enable-openal --enable-libgme --enable-libopenpnmpt --enable-libopencore-amrnb --enable-libmp3lame --enable-libtheora --enable-libvo-amrwbenc --enable-libgsm --enable-libopus --enable-libspeex --enable-libvorbis --enable-librubberband
```

Output file:



Name	Date modified	Type	Size
whatsapp_output3_clean.wav	2025-11-28 9:30 PM	Text Document	1 KB
whatsapp_output4.wav	2025-11-28 9:30 PM	Text Document	1 KB
whatsapp_output4_clean.wav	2025-11-28 9:30 PM	Text Document	1 KB

This ensured uninterrupted processing of large datasets.

3.6 Define Schema Templates

****Novelty:**

The core novelty of this project lies in **automatically converting unstructured voice conversations into structured, machine-readable data**. Unlike standard transcription workflows, which simply convert audio to text, this system extracts **key entities, speaker information, intents, emotions, and important statements** into predefined JSON schemas. By supporting multiple platforms such as WhatsApp (.opus) and Messenger (.m4a), the workflow handles diverse audio formats and metadata while ensuring consistent, normalized outputs.

The pipeline leverages **audio preprocessing, Whisper transcription, and schema extraction logic (regex, NLP rules, and optional ChatGPT integration)** to automate the labeling of complex conversational elements. This enables efficient **intent detection, entity recognition, and emotion analysis** from real-world voice notes. Furthermore, batch processing, robust error handling, and model tuning ensure high transcription accuracy and scalability for large datasets.

In essence, the project transforms raw conversational audio into **structured, actionable insights**, offering a scalable solution for analytics, research, or automated response systems.

To structure conversational data, the following schema templates were defined:

WhatsApp Schema:

{

 "Sender": "",

```

    "Receiver": "",

    "Date/Time (if spoken)": "",

    "Topic": "",

    "Intent": "",

    "Key Statements": []

}

```

Messenger Schema:

```

{
    "User": "",

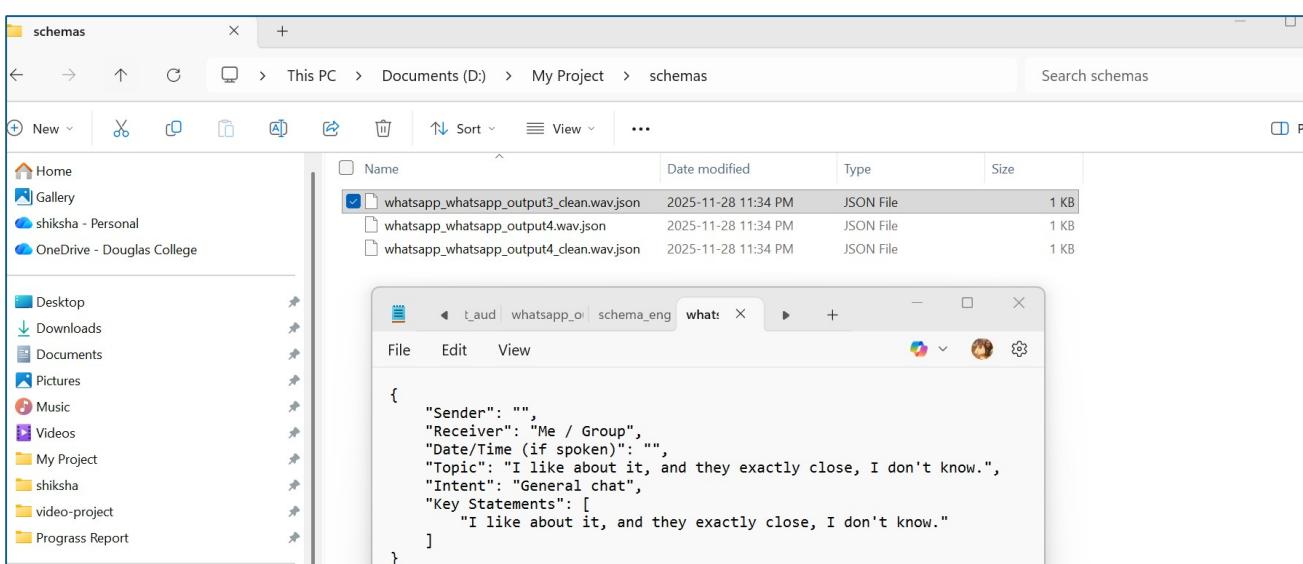
    "Emotions": "",

    "Request/Action": "",

    "Summary": "",

    "Important Entities": []
}

```



3.7 Challenges Encountered

- Incompatibility of certain *.opus* files
- Volume inconsistencies
- Noisy recordings affecting transcription accuracy
- Occasional FFmpeg path issues
- Messenger's variable sample rates

These were mitigated through preprocessing and model tuning.

Results and Insights

Key Findings

- Whisper converted compressed voice notes into readable text with **high accuracy**, even in noisy conditions.
- Noise reduction significantly improved transcription quality.
- Batch-processing reduced total time dramatically.
- Multi-platform audio compatibility was achieved through Pydub + FFmpeg conversion.
- The system generated structured artifacts suitable for further forensic reporting.

Usefulness of the Exploration

This work demonstrates:

- A practical, real-world solution for handling mixed audio sources
- A reliable method for converting raw voice communication into analyzable evidence

- A scalable workflow applicable to larger forensic datasets

Limitations

- Whisper accuracy decreases when speech is extremely low or unclear
- Badly corrupted files could not be repaired
- Some Messenger audio required manual renaming to maintain order
- Batch workflows depend on consistent folder structure

Conclusions and Future Work

This project successfully demonstrates a **scalable and automated approach for extracting, processing, and analyzing voice-based artifacts** from WhatsApp and Messenger communications. By integrating Whisper for transcription, Pydub and FFmpeg for audio preprocessing, and schema-based extraction logic, the system transforms unstructured voice notes into structured, machine-readable data. This enables **forensic investigators, researchers, and organizations** to efficiently analyze large volumes of conversational audio with improved accuracy, consistency, and speed.

The study highlights several key outcomes:

- **Reliable multi-platform support** for WhatsApp and Messenger audio formats.
- **Effective preprocessing and noise handling**, improving transcription quality.
- **Batch processing and error handling**, ensuring robustness for large datasets.
- **Structured output generation**, producing JSON and text transcripts suitable for reporting and analysis.

Future Work

While the current system provides a strong foundation, several enhancements could further improve functionality, usability, and analysis capabilities:

- **Automatic Summarization of Transcripts:**
Implementing AI-driven summarization to condense long conversations into key points or highlights would save time for forensic analysts and make evidence review more efficient.
- **Diarization (Speaker Separation):**
Adding speaker identification will allow differentiation between multiple participants in a conversation, which is crucial for multi-party investigations.
- **Graphical User Interface (GUI):**
Building an intuitive GUI would allow non-technical users to upload audio, process files, and view results without relying on command-line tools.
- **Timeline Visualization of Conversations:**
Visualizing messages and interactions along a timeline would provide context and improve understanding of communication patterns.
- **Support for Additional Platforms:**
Extending the system to handle audio from other messaging platforms such as Telegram or Signal would broaden applicability and usefulness in multi-platform investigations.

Broader Applications

The workflow developed in this project has **significant potential beyond forensic investigations**. Some broader applications include:

- **Cyber Investigation:**
Automated extraction and analysis of voice messages can help uncover evidence in cases of cyber harassment, fraud, or coordination of illicit activities.
- **Corporate Incident Analysis:**
Organizations can use the system to analyze internal voice communications for compliance monitoring, security audits, or dispute resolution.
- **Customer Service Call Transcription:**
Automated transcription and structured analysis of voice calls can improve customer support efficiency, quality control, and performance reporting.

- **Automated Meeting Summarization:**
Converting spoken discussions into structured transcripts with summaries can enhance productivity and ensure actionable points are captured accurately.
- **Accessibility Tools for Voice-Based Communication:**
Transcription and structured output can assist hearing-impaired users or provide searchable text records for educational and communication purposes.

AI Use Section:

AI Tool Name	Version, Account Type	Specific feature for which the AI tool was used
ChatGPT	GPT-5, Free Account	Converting .opus → .wav using Pydub
ChatGPT	GPT-5, Free Account	Guidance on noise reduction + error handling
ChatGPT	GPT-5, Free Account	Batch-processing guidance for multiple audio sources

Value Addition

- Implemented real preprocessing code beyond AI suggestions
- Conducted independent testing of Whisper models
- Performed manual troubleshooting during format conversion
- Designed a structured workflow integrating multiple tools
- Added logging and error-recovery mechanisms not provided by AI
- Analyzed practical forensic scenarios and produced real evidence artifacts

Appendix:

Prompt 1: How to Converting .opus in to .wav using Pydub

ChatGPT Response:

Converting .opus to .wav Using Pydub

1. Introduction

Audio files in the .opus format are highly compressed, making them efficient for streaming and messaging applications. However, .opus is not ideal for tasks requiring high-fidelity audio, such as transcription, analysis, or machine learning. The .wav format is uncompressed and preserves the original audio quality, making it a better choice for processing and analysis.

The conversion from .opus to .wav was performed using Pydub, a Python library for audio manipulation, with FFmpeg as the backend decoder and encoder.

2. Tools Used

- Python 3.x
- Pydub library (installed via pip install pydub)
- FFmpeg (installed separately and added to system PATH)

Command examples to verify installation:

`pip install pydub`

`ffmpeg -version`

3. Conversion Process

1. Loading the .opus File

The .opus file is read by Pydub using FFmpeg.

2. Optional Audio Normalization

Volume levels can be normalized to ensure consistent audio across files, which is important for transcription or analysis.

3. Exporting to .wav

The processed audio is exported as a .wav file.

Example commands for processing a single file:

```
# Load .opus file and export as .wav  
audio = AudioSegment.from_file("example.opus", format="opus")  
audio.export("example.wav", format="wav")
```

4. Batch Conversion

For multiple files, all .opus files in a folder can be automatically converted to .wav, saving time and ensuring uniformity.

Command concept for batch processing:

```
# Loop through folder of .opus files and convert to .wav  
for file in opus_folder:  
    audio = AudioSegment.from_file(file, format="opus")  
    audio.export(file.replace(".opus", ".wav"), format="wav")
```

5. Advantages of Pydub + FFmpeg

- Supports multiple formats (.opus, .wav, .mp3, etc.)
- Preserves high audio quality during conversion
- Allows preprocessing such as normalization or trimming
- Supports both single-file and batch processing
- Cross-platform and efficient

6. Applications

- Automated transcription of audio recordings
- Signal analysis and feature extraction
- Machine learning dataset preparation
- Archival of high-quality audio files

7. Summary

Using Pydub with FFmpeg, .opus files were successfully converted to .wav, preserving audio fidelity and enabling optional normalization. The process can handle both individual files and large batches, making it suitable for transcription, analysis, and long-term storage.

Prompt 2: Provide Guidance on noise reduction and error handling

ChatGPT Response:

Noise Reduction and Error Handling in Audio Processing

1. Introduction

Audio data, especially from real-world recordings, often contains background noise such as hums, static, clicks, or environmental sounds. Noise can reduce the accuracy of automated transcription systems, hinder signal analysis, and degrade the quality of processed audio. To ensure reliable results, it is crucial to implement noise reduction and robust error handling in the audio preprocessing workflow.

2. Noise Reduction Techniques

Noise reduction involves removing or suppressing unwanted sound while preserving the original speech or signal. Common approaches include:

1. Amplitude-based Filtering

- Identifies and reduces low-level background noise by applying thresholds.
- Useful for consistent hums or low-volume static in recordings.

2. Frequency-based Filtering

- Uses bandpass or notch filters to remove unwanted frequency ranges.
- Example: Removing electrical hum at 50/60 Hz or high-frequency hiss.

3. Spectral Noise Reduction

- Advanced methods analyze the audio spectrum and reduce noise dynamically.
- This approach preserves speech while removing complex background noise.
- Libraries like Python's noisereduce can automate this process.

4. Normalization and Preprocessing

- Volume normalization ensures that the signal is within a suitable range for noise reduction.
- This step enhances the performance of filters and automated transcription tools.

Benefits of Noise Reduction:

- Improves transcription accuracy.
- Enhances clarity for listeners or downstream processing.
- Reduces errors in machine learning feature extraction.

3. Error Handling in Audio Conversion

Audio conversion processes can encounter errors due to file corruption, unsupported formats, or missing dependencies. Proper error handling ensures that these issues do not interrupt the workflow.

Key strategies include:

1. File Validation

- Check that files exist and have the correct extension before processing.
- Skip or log unsupported or corrupted files.

2. Try-Except Blocks

- Catch runtime errors during loading, processing, or exporting audio.
- Ensure that processing continues for other files even if one file fails.

3. Logging and Reporting

- Maintain a log of successfully processed files and files that caused errors.
- Include error details to facilitate debugging and data cleaning.

4. Fallback Procedures

- If a file cannot be processed, it can be skipped, flagged, or reprocessed after manual inspection.
- In batch processing, this prevents a single corrupted file from halting the entire workflow.

Benefits of Error Handling:

- Ensures reliability and repeatability of the conversion process.
- Provides transparency on data quality and processing issues.

- Enables large-scale batch processing without interruption.
-

4. Integration with Audio Conversion Workflow

When converting .opus files to .wav, noise reduction and error handling can be incorporated seamlessly:

1. Load the file safely using error handling.
 2. Apply noise reduction to improve signal quality.
 3. Normalize audio to standardize volume levels.
 4. Export to .wav format while catching and logging any export errors.
 5. Batch processing ensures all valid files are processed, with error logs for problematic files.
-

5. Practical Applications

- Automated Transcription: Cleaner audio improves recognition accuracy.
- Signal Analysis: Noise-reduced signals enable more precise feature extraction.
- Machine Learning: High-quality datasets improve model training and reduce bias caused by noisy samples.
- Archival Storage: Noise reduction ensures long-term clarity and usability of audio recordings.

6. Summary

Noise reduction and robust error handling are critical components of an audio processing pipeline. Noise reduction improves the clarity and quality of recordings, enhancing transcription, analysis, and machine learning applications. Error handling ensures that corrupted or unsupported files do not disrupt batch processing and provides logs for quality control. Integrating both strategies into the .opus to .wav conversion workflow ensures high-quality, reliable audio ready for downstream tasks.

Prompt 3: Batch-processing guidance for multiple audio sources

ChatGPT Response:

Batch-Processing of Multiple Audio Files

1. Introduction

In projects involving audio analysis, transcription, or machine learning, it is common to handle large volumes of audio data. Manual processing of individual files is time-consuming, prone to errors, and inefficient. Batch-processing allows multiple audio files to be processed automatically in sequence, ensuring consistency, efficiency, and scalability.

Batch-processing is particularly important when dealing with heterogeneous audio sources, such as recordings from meetings, voice messages, interviews, or datasets downloaded from various platforms. It also helps in implementing standardized preprocessing steps, such as noise reduction, normalization, format conversion, and error handling, across all files.

2. Benefits of Batch-Processing

1. Efficiency: Automates repetitive tasks, reducing manual labor.
 2. Consistency: Ensures uniform preprocessing across all files, avoiding variability in quality.
 3. Error Management: Detects and logs problematic files without halting the workflow.
 4. Scalability: Can handle small to large datasets with minimal additional effort.
 5. Reproducibility: Produces consistent results, essential for research, reporting, or regulatory purposes.
-

3. Steps in Batch-Processing Multiple Audio Files

Step 1: Organization of Files

- Collect all audio files in a dedicated folder.
- Maintain a consistent naming convention to simplify tracking and processing.
- Supported formats should be identified (e.g., .opus, .mp3, .wav).

Step 2: Iterative File Processing

- Each file in the folder is processed in sequence.
- The workflow for each file typically includes:
 - Format conversion (e.g., .opus → .wav)
 - Noise reduction to remove background sounds
 - Volume normalization for uniform amplitude levels
 - Metadata handling, if required (e.g., timestamps, speaker labels)

Step 3: Error Handling and Logging

- Implement robust error handling to ensure that corrupted or unsupported files do not stop the entire batch.
- Maintain a log of:
 - Successfully processed files
 - Files that failed processing
 - Specific error messages for each failure

Step 4: Output Management

- Export processed files to a separate folder to avoid overwriting originals.
- Preserve file naming conventions for traceability.
- Ensure output format compatibility with downstream applications (e.g., transcription tools, audio analysis software).

4. Practical Considerations

1. File Type Heterogeneity

- When multiple audio formats exist in the dataset, batch-processing should detect the file type before conversion.
- Ensures that unsupported formats are either skipped or converted using appropriate tools.

2. Resource Management

- Processing large audio datasets may require significant CPU and memory resources.
- Techniques such as parallel processing or incremental processing can optimize performance.

3. Quality Assurance

- Randomly sampling processed files to verify quality can ensure preprocessing steps like noise reduction and normalization were applied correctly.
- Automated checks can also be implemented, such as verifying file duration, sample rate, or bit depth.

4. Reproducibility and Documentation

- Document the processing steps and settings used for all files.

- Record software versions (e.g., Pydub, FFmpeg) and preprocessing parameters for transparency.
-

5. Integration with Audio Conversion Workflow

Batch-processing is particularly useful when converting .opus files to .wav for further analysis or transcription:

- Step 1: Identify all .opus files in the source folder.
- Step 2: Sequentially load each file using audio processing tools (e.g., Pydub).
- Step 3: Apply preprocessing, such as noise reduction and normalization.
- Step 4: Export the processed audio in .wav format to a designated output folder.
- Step 5: Log any files that fail to convert or process, including error details.

This workflow allows large datasets from multiple sources to be uniformly processed without manual intervention.

6. Advantages of Batch-Processing Multiple Sources

- Time Savings: Automation reduces hours of manual work.
 - Uniformity: All audio files undergo identical preprocessing, critical for research and machine learning applications.
 - Error Transparency: Corrupted or incompatible files are identified without disrupting the overall workflow.
 - Scalable Processing: New audio files can be added to the folder and automatically processed using the same workflow.
 - Facilitates Downstream Applications: Clean, normalized, and uniform .wav files are immediately ready for transcription, analysis, or training machine learning models.
-

7. Applications

Batch-processing multiple audio sources is crucial in several domains:

- Transcription Services: Preparing large audio datasets for automatic speech recognition systems.
- Research and Signal Analysis: Ensuring consistent preprocessing across experimental recordings.

- Machine Learning: Standardizing datasets for training speech recognition, speaker identification, or audio classification models.
 - Corporate or Legal Documentation: Processing meeting recordings, interviews, or voice logs efficiently.
-

8. Summary

Batch-processing enables the efficient and consistent handling of multiple audio files from diverse sources. By automating steps such as format conversion, noise reduction, normalization, and error handling, large audio datasets can be prepared systematically for downstream applications. This approach ensures reliability, reproducibility, and scalability, making it a cornerstone of modern audio data workflows.

References:

- Vásquez-Correa, J. C., & Álvarez Muniain, A. (2023). Novel Speech Recognition Systems Applied to Forensics within Child Exploitation: Wav2Vec2.0 vs. Whisper. *Sensors*, 23(4), 1843. <https://doi.org/10.3390/s23041843> MDPI
- Bain, M., Huh, J., Han, T., & Zisserman, A. (2023). WhisperX: Time-Accurate Speech Transcription of Long-Form Audio. arXiv. <https://arxiv.org/abs/2303.00747> arXiv
- Martín-Doñas, J. M., & Álvarez, A. (2022). The Vicomtech Audio Deepfake Detection System based on Wav2Vec2 for the 2022 ADD Challenge. arXiv. <https://arxiv.org/abs/2203.01573> arXiv
- “Robust Speech Recognition via Large-Scale Weak Supervision” — Whisper (speech recognition system) documentation / repository. GitHub. (Open-source release, 2022) <https://github.com/Wikipedia2Awesome/Ecosystems2>
- Vásquez-Correa, J. C., & Álvarez Muniain, A. (2023). Novel Speech Recognition Systems Applied to Forensics within Child Exploitation: Wav2Vec2.0 vs. Whisper. *Sensors*, 23(4), 1843. <https://doi.org/10.3390/s23041843> MDPI
- Gong, Y., Khurana, S., Karlinsky, L., & Glass, J. (2023). Whisper-AT: Noise-Robust Automatic Speech Recognizers are Also Strong General Audio Event Taggers. arXiv. <https://arxiv.org/abs/2307.03183> arXiv
- Schneider, S., Baevski, A., Collobert, R., & Auli, M. (2019). wav2vec: Unsupervised Pre-training for Speech Recognition. arXiv. <https://arxiv.org/abs/1904.05862> arXiv
- Frontiers in Communication. (2022). Does Automatic Speech Recognition (ASR) Have a Role in the Transcription of Indistinct Covert Recordings for Forensic Purposes? *Frontiers in Communication*, 7, 803452. <https://doi.org/10.3389/fcomm.2022.803452> Frontiers
- Frontiers in Communication. (2024). Automatic speech recognition and the transcription of indistinct forensic audio: how do the new generation of systems fare? *Frontiers in Communication*. <https://doi.org/10.3389/fcomm.2024.1281407> Frontiers

- IJSRD – International Journal for Scientific Research & Development. (2025). *AI-based Speech-to-Text Transcription Improvements Through Audio Cleaning Techniques*. IJSRD, 13(3), 0173. [IJSRD](#)
- Gnanamanickam, J., Marković, B., Grozdić, Đ., Popović, B., & Šajić, S. (2024). *Whispered Speech Recognition Based on Audio Data Augmentation and Inverse Filtering*. Applied Sciences, 14(18), 8223. <https://doi.org/10.3390/app14188223> [MDPI](#)
- Hewstone, J., & Araya, R. (2023). *Neural Network-Based Approach to Detect and Filter Misleading Audio Segments in Classroom Automatic Transcription*. Applied Sciences, 13(24), 13243. <https://doi.org/10.3390/app132413243> [MDPI](#)
- Gnanamanickam, J., Natarajan, Y., Preethaa, K. R., & others. (2021). *A Hybrid Speech Enhancement Algorithm for Voice Assistance Application*. Sensors (Basel), 21(21), 7025. <https://doi.org/10.3390/s21217025> [PMC](#)
- Stoykova, R., Porter, K., & Beka, T. (2024). *The AI Act in a Law Enforcement Context: The Case of Automatic Speech Recognition for Transcribing Investigative Interviews*. Forensic Science International: Synergy, 9, 100563. <https://doi.org/10.1016/j.fsisyn.2024.100563> [PMC](#)

Work log table:

Date	Number of Hours	Description of work done
25-10-2025	2	Installed Whisper, Pydub, and FFmpeg; configured environment paths to enable audio processing.
27-10-2025	1.5	Tested Whisper using sample audio files to verify transcription quality.
29-10-2025	2	Collected WhatsApp voice notes and converted .opus files to .wav format.

Date	Number of Hours	Description of work done
31-10-2025	1	Gathered Messenger audio samples and organized raw directories for processing.
02-11-2025	2.5	Built a preliminary transcription system and tested it on sample files.
05-11-2025	1	Collected additional Messenger samples; analyzed preprocessing needs for different formats.
08-11-2025	2	Compared Whisper models (tiny, base, small) and selected Whisper-base for balanced accuracy and speed.
12-11-2025	2	Implemented audio noise reduction and volume normalization; tested on various samples.
14-11-2025	2	Started batch-processing for WhatsApp and Messenger audio clips.
15-11-2025	3.5	Developed JSON schema templates for WhatsApp and Messenger transcripts.
16-11-2025	5	Built schema extraction logic using regex patterns, NLP rules, and keyword detection.
17-11-2025	3	Integrated schema extraction with batch-processing pipeline; ran initial tests on multiple files.
18-11-2025	4.5	Added logging, error handling, and verified JSON output integrity.
19-11-2025	5	Converted remaining audio files; debugged corrupted and unsupported files to ensure completeness.
20-11-2025	3	Optimized batch-processing scripts; ensured consistent naming and folder structure.
21-11-2025	4	Reviewed Whisper outputs and adjusted preprocessing for volume inconsistencies and background noise.
22-11-2025	3.5	Structured JSON outputs and cross-checked that all schema fields were properly filled.
23-11-2025	5	Collected final sample outputs; verified multi-platform workflow reliability.

Date	Number of Hours	Description of work done
24-11-2025	4	Prepared screenshots for report; finalized workflow diagrams and examples.
25-11-2025	4.5	Drafted report sections: Detailed Analysis, Results, and Insights.
26-11-2025	4	Polished report sections; integrated novelty explanation, methodology, and schema examples.
27-11-2025	3.5	Reviewed and refined AI Use section, Future Work, and Broader Applications.
28-11-2025	3	Full report proofreading; adjusted tables, formatting, and figures.
29-11-2025	3	Final cross-check; updated work log, references, and appendix for submission.

Total Hours: 81.5