1) 内存 Dump 模板（String 字段 / JWT 定位）

作用

- `dump_class_string_fields`：指定一个类名，枚举该类的所有**活跃实例**，把实例里所有 `String` 字段（含私有字段）打印出来。用于找明文密码、JWT、临时密钥等。
- `search_jwt_in_static_strings`：遍历**已加载类**的**静态 String 常量**，用正则匹配 JWT-like 字符串并打印。用于快速定位硬编码/缓存的 token。

实现方式

- 都基于 Frida 的 Java 层 API。
- `dump_class_string_fields`：
  - `Java.choose("{class_name}", { onMatch })` 枚举 Java 堆上该类的所有存活对象；
  - 反射：`getDeclaredFields()` + `setAccessible(true)`；
  - 仅打印 `java.lang.String` 类型：`f.getType().getName() === 'java.lang.String'`；
  - 将 `f.get(instance)` 的值输出到 stdout → 后端通过 Socket.IO 实时推到前端控制台。
- `search_jwt_in_static_strings`：
  - `Java.enumerateLoadedClassesSync()` 拿到所有类名；
  - 反射遍历字段，配合 `java.lang.reflect.Modifier.isStatic(...)` 过滤静态字段；
  - 仅取 `String` 类型并读取 `f.get(null)`；
  - 用 JWT 正则（`xxx.yyy.zzz`）测试，命中则打印。

> 优势：开箱即用、低侵入；
> 限制：需要类已加载/对象存在；遍历规模大时可能慢，推荐在目标 App 已进入目标页面/逻辑后再执行。

---

2) SSL Pinning Bypass 模板

作用

在测试/CTF 环境中动态**绕过证书校验**，让你能用自签证书/代理抓 HTTPS 明文（便于配合 mitmproxy 做联动分析）。

实现方式

- Frida 注入后：
  - 定义 `TrustAllManager` 实现 `X509TrustManager`（三个方法空实现）；
  - Hook `javax.net.ssl.SSLContext.init(KeyManager[], TrustManager[], SecureRandom)`：
    - 把传入的 `TrustManager[]` 替换为 `[TrustAllManager]`；
    - 输出 “bypass” 日志到 stdout。

- 运行链路：前端点“执行模板”→ `/api/run_frida` → 写入临时脚本 → `frida -U (-n|-f)` 启动 → `_emit_frida_output` 将 stdout/stderr 通过 Socket.IO 推给前端。

> 说明：该模板**只放宽服务器证书校验**。如果目标是**双向 TLS（需要客户端证书）**，仍需要额外处理 `KeyManager` 或应用侧逻辑，模板本身并不“伪造客户端证书”。

---

3) SharedPreferences Dump 模板

作用

拦截 Android `SharedPreferences` 的 `getString(key, def)` 调用，**在每次读取时**打印当前读取的键和值。常用于快速看到 App 取出的 token、会话 id、配置密钥等。

实现方式

- Hook `android.app.SharedPreferencesImpl.getString(String, String)`；
- 在实现里 `console.log("[SharedPreferences] " + key + " = " + value)`；
- 仅在**被读取时**打印（不会做全量导出）；配合 App 的正常操作更容易触发命中。

---

4) Frida 运行链路升级（更灵活的三种入口）

作用

你可以用**三种方式**跑 Frida 脚本，适配不同场景：

1. **直接传脚本字符串**：`POST /api/run_frida` 时带 `script` 字段；
2. **按模板+参数运行**：传 `template` 和 `template_params`（如 `{ class_name: '...' }`）；
3. **基于会话自动生成**：传 `id`（某条 MITM 会话），后端按有无 body 自动套用 `requestbody_dump` 或 `okhttp_log_url`。

实现方式（后端）

- `/api/generate_frida`：支持 `template + template_params` 的渲染；否则回落到“基于 session 自动选择模板”的旧逻辑。
- `/api/run_frida`：
  - 优先级：`script` > (`template` + `params`) > `session id` > 默认；
  - 将最终脚本写到 `FRIDA_DIR`；
  - 通过 `subprocess.Popen(["frida", "-U", ...])` 启动（附加到进程 `-n` 或 spawn `-f --no-pause`）；
  - `stdout/stderr` 用后台线程 `_emit_frida_output` 逐行读取并 `socketio.emit("frida_log", ...)` 实时推送；
  - 同时发送一次 `frida_started` 事件，前端能看到 PID 和脚本路径。

> 安全控制：若设置了 `ADMIN_TOKEN` 环境变量，`/api/run_frida`、`/api/clear_sessions` 等接口必须带 `X-ADMIN-TOKEN`。

---

5) 前端“Memory / Frida Templates”操作区

作用

在同一页面上，**既能看抓包会话**，又能**一键运行 Frida 模板**并看实时输出，串起“流量 ↔ 内存/Hook”的联动闭环。

实现方式（前端）

- 新增模板选择下拉框 + 类名输入框 + “执行模板”按钮：
  - 选择 `dump_class_string_fields` 时提示输入类名；
  - 点击执行：弹窗询问包名、（可选）管理员 Token → 组装 `template` / `template_params` 调用 `/api/run_frida`；
- Socket.IO 客户端订阅：
  - `session_new`：mitmproxy 新会话到达时刷新列表；
  - `frida_log`：控制台区域实时打印 Frida 输出；
  - `frida_started`：显示 PID/脚本路径。

---

6) 会话展示保持原样（兼容 mitmproxy 抓包）

作用

不影响你原来的抓包链路：mitmproxy 脚本仍向 `/api/sessions` 上报，前端表格实时看到新会话、点行可查看详情，也能导出/清空。

实现方式

- `/api/sessions` `POST/GET`、`/api/sessions/<id>`、`/api/sessions/export`、`/api/clear_sessions` 全保留；
- 新会话存入 SQLite 后，后端 `socketio.emit("session_new", {...})` 通知前端刷新；
- 搜索框（回车触发）仍可按 URL、方法、body 子串检索（下一步我们可以做成“高级过滤”）。