1) 总览 & 进展对比（两句话开场）

- 上次进展：**前端可视化完成**；**Frida 注入（模板/自定义）尚未可用。**
- 本次进展：**模板化注入（SSL Pinning Bypass、SharedPreferences Dump 等）+ 自定义注入均已跑通**，注入日志在前端 Console 实时回显；端到端闭环（mitm → Flask → 前端 → Frida 注入 → 回显）完成。前端模板入口与交互见页面下拉选项与按钮（`ssl_pinning_bypass` / `sharedprefs_dump` /自定义脚本区）

小白总结版：以前只能看流量，现在能一键注入、现场看 Hook 日志了。

---

2) 系统结构 & 数据流（1 张图的口播版）

1. **设备/模拟器** → 设置系统代理到 mitmproxy
2. **mitmproxy 插件** `mitm_to_flask.py` → 抓到请求，做敏感字段检测，队列 + 重试节流，POST 给后端（可通过环境变量调节上报节奏/重试/超时）
3. **Flask 后端** `app.py` → 入库（SQLite）+ 通过 Socket.IO 广播到前端（会话列表/详情实时刷新）
4. **前端** `index.html` → 点“运行模板/自定义脚本”，后端生成脚本并调用 Frida，**Frida stdout** 通过 WebSocket **绿色控制台**实时显示（包含 `[WRAP]` 前缀的控制台包装日志）

小白总结版：抓包进后端、前端看详情，一键注入 Frida，日志立刻弹出来。

---

3) 各模块是怎么实现的？（逐一讲清“干了啥、怎么干、证据在哪”）

3.1 前端可视化（已完成）

- **会话列表/详情**：表格实时更新，会话详情区展示 URL/Method/Headers/Body，敏感字段高亮（结构与样式在 `index.html`）
- **Frida 模板与自定义脚本**：右侧面板下拉选择 `ssl_pinning_bypass` / `sharedprefs_dump` 等模板，或在“自定义脚本”文本框直接注入；有按钮“生成脚本 / 运行”与 Spawn 选项
- **Console 回显**：WebSocket 通道把后端转发的 Frida stdout 渲染在绿色控制台区域（高而可滚动）

小白总结版：界面能看流量，也能点按钮跑脚本，输出都在右侧绿色终端。

3.2 后端 Flask（核心中枢）

- **会话收集 & 广播**：`POST /api/sessions` 写 SQLite 并通过 Socket.IO 推给前端；查询/导出接口补齐管理闭环（README 有接口概览）
- **Frida 模板执行**：提供生成/运行 API，把选定模板渲染成脚本，再调用 `frida -U -n <包名> -s <脚本>`；stdout 每行包装后通过 WebSocket 推送（“控制台包装器”会输出 `[WRAP]`）
- **权限保护**：可选的 `ADMIN_TOKEN` 头校验保护敏感操作（README 指明；后端有装饰器）

小白总结版：后端负责“收/存/推”和“生成+执行 Frida 脚本”，并把日志喂给前端。

3.3 mitm 插件（稳定上报）

- **队列与重试节流**：请求入队，后台 worker 批量 POST，带最大重试次数、超时、最小发送间隔的保护；这些参数可用环境变量覆盖（例如 `MITM_POST_INTERVAL`、`MITM_MAX_RETRIES`）
- **目的**：保证高并发/弱网络下也能把会话稳定送达 Flask。

小白总结版：mitm 是“运货车”，抓到包就稳定送到后端。

3.4 Frida 注入（这次的核心亮点）

- **模板化注入**（举两例）：
  - **SharedPreferences Dump**：Hook `getString/Editor.put*` 等方法，打印 key/value；拿不到文件名时会显示 `(unknown-file)`，但键值会照样打印（你这次的回显与模板行为一致）
  - **SSL Pinning Bypass**：多点位覆盖，Hook `SSLContext.init`（替换 TrustManager）、`OkHostnameVerifier.verify`（直接返回 true）、`CertificatePinner.check / check$okhttp`（拦截并放行），回显会打 `[bypass]` 前缀
- **自定义注入**：前端“自定义脚本”Tab 允许直接写 Frida JS 注入，后端以相同机制运行并把 stdout 回传（按钮与文本域在页面里）
- **类加载器/实例枚举**：像 `dump_class_string_fields` 这类模板会**自动选择合适的 ClassLoader**（通过 `BuildConfig` 或枚举加载器），再 `Java.choose` 枚举出实例打印字段值（避免被多 Dex/插件化影响）

---

4) “Java 不可用（Java is not available）”老问题：原因 & 彻底解决方案

> 现象：注入时报错 *Java is not available* 或 hook 失败。

根因梳理（常见三类）

1. **时机不对**：附加过早、Java VM/ART 尚未就绪。
2. **ClassLoader 不匹配**：目标类并不在默认 Loader，`Java.use` 找不到。
3. **进程/包名不对**：没附到真正的 App 进程或附加到纯 native 进程。

你现在的解决组合拳

- **统一使用** `Java.perform`：所有模板在 `Java.perform(function(){ ... })` 内执行，保证 ART 可用。
- **自动选 Loader**：模板会先尝试**通过** `BuildConfig` **推断包名并枚举 ClassLoader**，命中即 `Java.classFactory.loader = loader`，再做 `Java.use` / `Java.choose`（代码里有这段逻辑）
- **必要时用 Spawn**：前端支持 **Spawn 模式**（复选框），确保从进程启动时即附加，避免“已运行但还没 Java 环境”的窗口期（UI 上有 Spawn 复选框）

  Probe 功能先探测再注入：前端“Probe(查找进程/类名)”按钮帮助定位可附加进程，减少附错进程导致的 “Java 不可用”

效果验证

- 你最近的 Console 日志能稳定出现 `[WRAP] console wrapper loaded`（脚本启动包装）与 `[bypass]` / `[SharedPreferences]` 等分支日志，说明 **Java.perform + ClassLoader 选择** 已经解决了“Java 不可用/类找不到”的关键问题；日志前缀和模板行为对得上（Loader 选中、调用点命中、回显正常）