

Android Reverse Engineering Cookbook

从入门到精通的 Android 逆向与安全分析

基础知识、工具指南、实战技巧与进阶主题

涵盖 Frida、Unidbg、Xposed、IDA Pro 等核心工具

包含数据分析、工程实践与真实案例研究

Authors: +5/Gemini Pro 3.0/Claude Code Opus 4.5

Email: overkazaf@gmail.com

WeChat: _0xAF_

 17 Created: 2025-08-01

 Last Revised: 2025-12-20

 Version: v2.0

写在前面

逆向工程，这门充满神秘色彩的技术艺术，一直是安全研究者、开发者和技术爱好者心中的“圣杯”。它不仅需要扎实的编程功底，更需要对系统底层的深刻理解，以及那份在迷宫中寻找出口的耐心与智慧。然而，市面上的逆向工程资料要么过于零散，要么晦涩难懂，让许多初学者望而却步。

本书的诞生，源于我近期阅读rOysue的多本逆向书籍后（如《Frida Android SO逆向深入实践》，《unidbg逆向工程原理与实践》等）的冲动，以及多年逆向实战中的一个朴素愿望：将散落在各处的知识碎片，系统地串联成一条清晰的学习路径。最初只是想整理平时的学习笔记，以便在旅途中或飞机上离线阅读与复习。而当AI浪潮以不可阻挡之势席卷而来，我意识到这已不再是“要不要拥抱”的问题，而是“如何更好地拥抱”——为什么不尝试一次真正意义上的人机协作呢？

于是，这本手册成为了一次独特的创作实验。在这个项目中，分工明确而高效：

- Claude Code Opus 4.5 化身“牛马程序员”，负责所有代码示例的编写与调试、大型代码库的深度理解与重构、架构流程图的创建、批量处理 Markdown 格式问题，以及自动化文档生成流程
- Gemini Pro 3.0 担当“科研老师傅”，负责海量技术知识点的调研、梳理与发散，从 arXiv 前沿论文到工业界实践资料的深度阅读与分析
- 而我则作为“Agent善后工程师”，负责全书的顶层架构设计、内容审核及关键技术把关，以及疯狂地指挥Agent更合理地消耗token

这本书适合谁？

无论你是安全研究员、渗透测试工程师、数据工程师，还是对底层技术充满好奇的开发者，本书都将为你提供从入门到进阶的完整路径。我们采用“配方式”的组织结构，每个章节都是一个独立的实战案例，你可以按需阅读，也可以系统学习。

书中涵盖了网络抓包、加密分析、反检测对抗、脱壳修复、动态分析等核心主题，并配有大量可直接运行的代码示例。我们的目标是：让你读完每一个配方后，都能立即动手实践。

推荐学习路径：

- 零基础入门：快速入门(Q) → 基础知识(F) → 工具指南(T) → 基础配方(R01-R15)
- 安全研究员：反检测(R06-R11) → 脱壳技术(R12-R15) → 分析技术(R16-R23) → 案例分析(C)
- 数据工程师：网络分析(R01-R05) → 自动化(R24-R31) → 工程实践(E) → 脚本集合(R32-R37)
- 进阶开发者：工具内部原理(T02/T04/T06) → 进阶主题(A) → 真实案例(C)

分类编号系统

本手册采用分类编号系统，每个章节都有一个唯一标识符，便于快速检索和引用：

Q - 快速入门	R - 实战配方	T - 工具指南
C - 案例分析	F - 基础知识	A - 进阶主题
E - 工程实践	X - 附录资源	

例如： R01 表示第一个实战配方， T05 表示第五个工具指南

关于本版本

您正在阅读的是静态 PDF 版本，专为离线阅读和打印优化。内容定期更新，适合系统性学习和随时查阅。

我们正在构建动态交互版本，支持与 AI Agent / LLM 实时对话，可以针对具体问题获得定制化解答、代码示例生成、以及更深入的互动学习体验。敬请期待！

目录

| 首页

| 快速入门

 | Q01 关于本书

 | Q02 快速入门指南

 | Q03 环境配置

| 实战配方

 | 网络分析

 | R01 网络抓包分析

 | R02 加密算法分析

 | R03 TLS 指纹识别

 | R04 JA3 指纹技术

 | R05 JA4 指纹技术

 | 反检测

 | R06 Frida 反调试绕过

 | R07 Xposed 反调试绕过

 | R08 验证码绕过技术

 | R09 应用加固识别

 | R10 设备指纹与绕过

R11 移动安全与反爬

脱壳技术

R12 脱壳概述

R13 Frida 脱壳与 SO 修复

R14 SO 混淆与反混淆

R15 SO 字符串解密

分析技术

R16 逆向工程流程

R17 静态分析深入

R18 动态分析深入

R19 OLLVM 反混淆

R20 VMP 虚拟机分析

R21 JS 混淆器分析

R22 JS 虚拟机保护

R23 Native 字符串混淆

自动化

R24 自动化与设备农场

R25 拨号代理池

R26 代理池设计

R27 Scrapy 爬虫框架

R28 Scrapy-Redis 分布式

R29 Docker 部署

R30 虚拟化与容器

R31 Web 反爬技术

脚本集合

- [R32 Frida 脚本示例](#)
- [R33 Frida 常用脚本](#)
- [R34 自动化脚本](#)
- [R35 Native Hook 技术](#)
- [R36 Objection 代码片段](#)
- [R37 C 语言仿真脚本](#)

Native 开发

- [R38 JNI 开发与调用原理](#)
- [R39 SO 动态库编译与加载](#)

工具指南

动态分析工具

- [T01 Frida 使用指南](#)
- [T02 Frida 内部原理](#)
- [T03 Xposed 使用指南](#)
- [T04 Xposed 内部原理](#)
- [T05 Unidbg 使用指南](#)
- [T06 Unidbg 内部原理](#)

静态分析工具

- [T07 Ghidra 入门指南](#)
- [T08 IDA Pro 入门指南](#)
- [T09 Radare2 入门指南](#)

速查手册

T10 ADB 命令速查

案例研究

C01 反分析技术案例

C02 音乐 App 案例

C03 社交媒体与风控案例

C04 App 加密签名案例

C05 视频 App 与 DRM 案例

C06 Unity 游戏 (Il2Cpp) 案例

C07 Flutter 应用案例

C08 恶意软件分析案例

参考资料

基础知识

F01 APK 文件结构

F02 Android 四大组件

F03 AndroidManifest 解析

F04 Android Studio 调试工具

F05 DEX 文件格式

F06 Smali 语法入门

F07 SO/ELF 文件格式

F08 ART 运行时

F09 ARM 汇编入门

F10 x86 与 ARM 汇编基础

TOTP 时间动态密码

进阶主题

- A01 Android 沙箱实现
- A02 AOSP 与系统定制
- A03 AOSP 深度改机
- A04 最小化 Android RootFS
- A05 SO 反调试与混淆
- A06 SO 运行时仿真

工程实践

- E01 框架与中间件
- E02 消息队列
- E03 Redis 常用命令
- E04 风控 SDK 编译指南

数据分析

- E05 数据仓库与处理
- E06 Apache Flink
- E07 HBase 数据库
- E08 Apache Hive
- E09 Apache Spark

附录

- X01 GitHub 开源项目
- X02 学习资源
- X03 CTF 练习平台

X04 术语表

配方编号系统

章节总数: 89

首页

快速入门

| [Q01] 关于本书

Q01: 关于本书

本书是一本 Android 逆向工程实战手册，采用 Cookbook（配方）形式组织——每个章节都是一个独立的“配方”，针对特定问题提供完整解决方案。

如何使用

- 遇到问题 → 在「实战配方」章节查找对应配方
- 学习工具 → 查阅「工具指南」章节
- 看实战案例 → 参考「案例研究」章节
- 补充基础 → 阅读「参考资料」章节

适合谁

读者类型	推荐路径
安全研究员	参考资料 → 工具指南 → 反检测 → 案例研究
爬虫开发者	网络分析 → 加密算法分析 → 自动化
逆向爱好者	快速入门 → 参考资料 → 实战配方

前置要求

- 熟悉至少一门编程语言（Python/Java/JS）
- 了解 Android 基础（Activity、Service 等）

-
- 会使用命令行

免责声明

本书仅供学习和安全研究。请确保你有权分析目标应用，并遵守相关法律法规。

准备好了？→ [10 分钟快速入门](#)

[Q02] 快速入门指南

Q02: 快速入门指南

欢迎！这个指南将帮助你在 10 分钟内完成第一次 Android 逆向分析。

你将学到什么

完成本指南后,你将能够:

- 在真机/模拟器上运行 Frida
- Hook 一个 Android 应用的 Java 方法
- 查看和修改方法的参数与返回值
- 理解基本的逆向分析流程

预计用时: 10-15 分钟

前置条件

必需工具

工具	说明
□ Android 设备	已 Root 的真机或模拟器(推荐 Genymotion / Android Studio AVD)
□ ADB	Android Debug Bridge
□ Python	版本 3.8+
□ 测试 App	本指南使用系统自带的设置应用

检查清单

```
# 1. Check if ADB is installed  
adb version  
  
# 2. Check if Python is installed  
python3 --version  
  
# 3. Check device connection  
adb devices  
# Should display your device
```

操作步骤

第 1 步: 安装 Frida (2 分钟)

在电脑上安装 Frida 工具:

```
pip install frida-tools
```

在 Android 设备上安装 frida-server:

```
# Visit https://github.com/frida/frida/releases
# Download frida-server matching your Python frida version

# View your frida version
frida --version

# View device architecture
adb shell getprop ro.product.cpu.abi
# Common output: arm64-v8a, armeabi-v7a, x86_64
```

```
# Decompress and push to device
unzip frida-server-*.zip
adb push frida-server-*-android-* /data/local/tmp/frida-server

# Grant execute permission and run
adb shell "chmod 755 /data/local/tmp/frida-server"
adb shell "/data/local/tmp/frida-server &"
```

验证安装:

```
frida-ps -U
# Should see output like:
# PID Name
# -----
# 1234 com.android.settings
# 5678 com.android.systemui
# ...
```

第 2 步: 编写第一个 Hook 脚本 (3 分钟)

我们将 Hook Android 设置应用,监控其方法调用。

创建 Hook 脚本 `first_hook.js`:

```
// first_hook.js - your first Frida script

Java.perform(function () {
    console.log("\n[*] Frida hook started!");
    console.log("[*] Finding TargetClass...\n");

    // Hook Android system Log class
    var Log = Java.use("android.util.Log");

    // Hook Log.d method (debug log)
    Log.d.overload("java.lang.String", "java.lang.String").implementation =
        function (tag, msg) {
            console.log("\n[+] Captured LogCall:");
            console.log("    Tag: " + tag);
            console.log("    Message: " + msg);

            // Call original method
            return this.d(tag, msg);
    };

    console.log("[*] Hook setup completed! Now open Settings app...\n");
});
```

运行 Hook 脚本:

```
# Method 1: attach to running app
frida -U -n com.android.settings -l first_hook.js

# Method 2: inject at app startup
frida -U -f com.android.settings -l first_hook.js --no-pause
```

预期输出:

```
[+] Captured LogCall:
  Tag: SettingsActivity
  Message: onCreate called

[+] Captured LogCall:
  Tag: SettingsFragment
  Message: Loading preferences...
```

✓ 如果看到类似上方的日志输出, 恭喜你已经成功 Hook 了一个 Android 应用!

第 3 步: 修改应用行为 (3 分钟)

现在让我们做点更有趣的——修改应用的返回值。

创建脚本 `modify_behavior.js` :

```
// modify_behavior.js - Modify App behavior

Java.perform(function () {
    console.log("\n[*] Start Hook...\n");

    // Hook String Class equals Method
    var String = Java.use("java.lang.String");

    String.equals.implementation = function (other) {
        // get original result
        var result = this.equals(other);

        // if string is "WiFi", modify result
        if (this.toString() === "WiFi" || other.toString() === "WiFi") {
            console.log("\n[!] detected WiFi String comparison");
            console.log(
                "    Original: '" + this + "' == '" + other + "' => " + result
            );
            console.log("    Modified: true\n");
            return true; // return true
        }

        return result;
    };

    console.log(
        "[*] Hook completed! All 'WiFi' String comparison will return true\n"
    );
});
```

运行脚本:

```
frida -U -f com.android.settings -l modify_behavior.js --no-pause
```

你可以用同样的方法:

- 修改加密参数
- 绕过签名验证

-
- 篡改网络请求
-

恭喜你，现在你已完成快速入门

学会了什么？

- 安装和运行 Frida
- 编写基本的 Hook 脚本
- 监控方法调用
- 修改方法返回值

下一步学习

根据你的兴趣选择：

深入学习工具

- [Frida 完整指南](#) - 学习 Frida 的所有 API
- [Frida 内部原理](#) - 理解 Frida 如何工作
- [ADB 命令速查](#) - 掌握 ADB 常用命令

解决具体问题

场景 1：抓包分析 → [网络抓包](#)

场景 2：绕过反调试 → [反调试绕过](#)

场景 3：分析加密算法 → [密码学分析](#)

场景 4：脱壳加固 App → [应用脱壳](#)

实战案例

- [音乐 App 分析](#) - VIP 破解、音频解密
-

- 社交 App 风控 - API 签名、设备指纹

理解基础原理

- APK 文件结构
- Android 四大组件
- DEX 文件格式



常见问题

Q: Frida 连接不上设备?

```
# 1. Confirm frida-server is running  
adb shell "ps | grep frida"  
  
# 2. Reboot frida-server  
adb shell "pkill frida-server"  
adb shell "/data/local/tmp/frida-server &"  
  
# 3. Check port forwarding (if needed)  
adb forward tcp:27042 tcp:27042
```

Q: Hook 不生效?

排查步骤:

1. 确认应用正在运行:

```
frida-ps -U | grep YourAppPackageName
```

1. 检查类名是否正确:

- 使用 jadx-gui 反编译查看准确的类名

- 注意内部类的 \$ 符号(如 OuterClass\$InnerClass)

1. 处理方法重载:

```
// If method has multiple overloads, need to specify parameter class type
YourClass.yourMethod.overload("java.lang.String").implementation = function (
    arg
) {
    // your code here
};
```

Q: 应用检测到 Frida?

→ 查看 [Frida 反调试绕过](#)

更多资源

项目	说明
Frida 官方文档	https://frida.re/docs/
Frida CodeShare	https://codeshare.frida.re/ (社区脚本)
本 Cookbook 脚本库	Frida 脚本示例

准备好开始你的逆向之旅吧!

[Q03] 环境配置

Q03: 环境配置

5 分钟完成基础逆向环境搭建。

必需工具

工具	安装命令
Python 3.8+	官网下载或 <code>brew install python3</code>
ADB	<code>brew install android-platform-tools</code> 或 Android Studio
Frida	<code>pip install frida-tools</code>

设备准备

推荐新手：使用模拟器（Genymotion 或 Android Studio AVD）

使用真机：

1. 设置 → 关于手机 → 连点“版本号”7 次 → 启用开发者选项
2. 开发者选项 → 启用 USB 调试
3. 连接设备，执行 `adb devices` 确认

安装 Frida Server

```
# 1. 查看设备架构  
adb shell getprop ro.product.cpu.abi  
# arm64-v8a → frida-server-*–android-arm64  
# x86_64 → frida-server-*–android-x86_64  
  
# 2. 下载对应版本（版本需与本地 frida 一致）  
# https://github.com/frida/frida/releases  
  
# 3. 部署到设备  
adb push frida-server /data/local/tmp/  
adb shell "chmod 755 /data/local/tmp/frida-server"  
adb shell "/data/local/tmp/frida-server &"  
  
# 4. 验证  
frida-ps -U
```

验证清单

```
python3 --version      # >= 3.8  
adb devices            # 显示设备  
frida --version        # 显示版本  
frida-ps -U           # 列出进程
```

环境就绪！ → [开始 10 分钟入门](#)

实战配方

| 网络分析

[R01] 网络抓包分析

R01: 抓包分析 Android 应用的网络流量

问题场景

- 你遇到了什么问题？
 - 想知道某个 App 调用了哪些 API 接口
 - 需要分析 API 的请求参数和响应数据
 - 想查看 App 发送了哪些敏感信息（设备信息、定位等）
 - 需要找到加密签名的生成逻辑
 - 想重放或修改 API 请求
 - 本配方教你：配置抓包环境，拦截并分析 HTTPS 流量，绕过 SSL Pinning 限制。
 - 预计用时：15-30 分钟（首次配置）
-

工具清单

必需工具

- Android 设备/模拟器（已 Root，或可安装证书）
 - 抓包代理工具（选择其一）：
 - Burp Suite（推荐，功能最强）
 - Charles（UI 友好）
 - mitmproxy（开源，可编程）
 - Frida（用于绕过 SSL Pinning）
-

可选工具

- Wireshark (分析底层 TCP/UDP 流量)
- HttpCanary (Android 上的抓包工具, 无需 PC)

前置条件

确认清单

```
# 1. Verify device connection  
adb devices  
  
# 2. Frida 可用  
frida-ps -U  
  
# 3. PC and phone on the same Wi-Fi network  
# Record PC IP address (used below as YOUR_PC_IP)  
# Windows: ipconfig  
# macOS/Linux: ifconfig or ip addr
```

- Android 7.0+: 需要 Root 权限安装系统证书
- Android 6.0-: 可直接安装用户证书, 无需 Root
- 或使用支持用户证书的 App (Target SDK < 24)

解决方案

第 1 步：配置抓包工具（5 分钟）

使用 Burp Suite (推荐)

1.1 启动 Burp Suite

```
# Download Burp Suite Community Edition (free)
# https://portswigger.net/burp/communitydownload

# Run the command
java -jar burpsuite_community.jar
```

1. 打开 Proxy → Options

2. 在 Proxy Listeners 部分

3. 点击 Add, 配置:

- Bind to port: 8888
- Bind to address: All interfaces (或选择你的 Wi-Fi 网卡)

1. 点击 OK 保存

Burp Proxy配置

验证: 浏览器访问 http://YOUR_PC_IP:8888, 应该看到 Burp 的错误页面 (表示代理工作正常)

使用 Charles

1.1 启动 Charles

下载: <https://www.charlesproxy.com/download/>

1.2 配置代理

- Proxy → Proxy Settings
- 设置 Port 为 8888
- 勾选 Enable transparent HTTP proxying

使用 mitmproxy

```
# Install  
pip install mitmproxy  
  
# startup (监听 8888 端口)  
mitmproxy -p 8888 --listen-host 0.0.0.0  
  
# 或使用 Web 界面  
mitmweb -p 8888 --listen-host 0.0.0.0  
# 访问 http://127.0.0.1:8081 查看流量
```

第 2 步：配置手机代理（2 分钟）

2.1 连接到同一 Wi-Fi

确保手机和 PC 在同一局域网。

2.2 设置手动代理

1. 打开手机 设置 → Wi-Fi

2. 长按当前连接的 Wi-Fi → 修改网络

3. 展开 高级选项

4. 代理设置改为 手动：

- 代理服务器主机名: YOUR_PC_IP (如 192.168.1.100)

- 代理服务器端口: 8888

1. 保存

2.3 验证代理连接

```
# 手机浏览器访问任意 HTTP 网站 (如 http://example.com)  
# 此时 Burp/Charles 应该显示拦截到的请求
```

第 3 步：安装 HTTPS 证书（5-10 分钟）

为什么需要？HTTPS 流量经过加密，需要安装证书才能解密查看。

Burp Suite 证书安装

3.1 下载证书

1. 手机浏览器访问 `http://burp`
2. 点击 CA Certificate 下载 `cacert.der`

3.2 安装证书

- Android 7.0+（需要 Root）：

```
# 1. 转换证书格式
openssl x509 -inform DER -in cacert.der -out cacert.pem

# 2. 计算证书哈希
HASH=$(openssl x509 -inform PEM -subject_hash_old -in cacert.pem | head -1)

# 3. 重命名并推送到系统目录
cp cacert.pem $HASH.0
adb root
adb remount
adb push $HASH.0 /system/etc/security/cacerts/
adb shell chmod 644 /system/etc/security/cacerts/$HASH.0

# 4. 重启设备
adb reboot
```

Charles 证书安装

1. 手机浏览器访问 `http://chls.pro/ssl`
2. 下载并安装证书
3. Android 7.0+ 同样需要安装到系统目录（参考 Burp 步骤）

mitmproxy 证书安装

1. 手机浏览器访问 `http://mitm.it`
2. 点击 Android 图标下载证书

3. 安装步骤同上

第 4 步：开始抓包（1 分钟）

4.1 清空旧记录

- Burp: Proxy → HTTP history → 右键 → Clear history
- Charles: Proxy → Clear Session

4.2 启动目标 App

在手机上打开要分析的应用，正常使用。

4.3 查看流量

在抓包工具中：

- 查看 HTTP history / Sequence
- 筛选目标 App 的域名
- 分析 Request/Response 内容

示例分析点：

- 请求 URL 和参数
 - Request Headers（User-Agent, Authorization, 自定义签名头）
 - Request Body（POST 数据）
 - Response Body（API 返回的 JSON/XML）
-

第 5 步：绕过 SSL Pinning（如遇到）

症状：

- 证书已安装，但 HTTPS 请求仍无法抓取
- App 显示“网络错误”或直接闪退
- 抓包工具显示 SSL 握手失败

原因：App 启用了 SSL Pinning（证书锁定），拒绝信任系统证书。

方法 1：使用 Frida 通用脚本（推荐）

下载脚本 `bypass_ssl_pinning.js`：

```
// Universal android SSL Pinning Bypass
Java.perform(function () {
    console.log(" [SSL Pinning Bypass] 已启动");

    // 拦截 TrustManagerImpl (常用)
    try {
        var TrustManagerImpl = Java.use(
            "com.android.org.conscrypt.TrustManagerImpl"
        );
        TrustManagerImpl.verifyChain.implementation = function (
            untrustedChain,
            trustAnchorChain,
            host,
            clientAuth,
            ocspData,
            tlsSctData
        ) {
            console.log("✓ [TrustManagerImpl] BypassCertValidate: " + host);
            return untrustedChain;
        };
    } catch (e) {
        console.log("! TrustManagerImpl 不存在");
    }

    // Hook OkHttp3
    try {
        var CertificatePinner = Java.use("okhttp3.CertificatePinner");
        CertificatePinner.check.overload(
            "java.lang.String",
            "java.util.List"
        ).implementation = function (hostname, peerCertificates) {
            console.log("✓ [OkHttp3] Bypass SSL Pinning: " + hostname);
            return;
        };
    } catch (e) {
        console.log("! OkHttp3 不存在");
    }

    // Hook SSLContext
    try {
        var SSLContext = Java.use("javax.net.ssl.SSLContext");
        SSLContext.init.overload(
            "[Ljavax.net.ssl.KeyManager;",
            "[Ljavax.net.ssl.TrustManager;",
            "java.security.SecureRandom"
        ).implementation = function (keyManager, trustManager, secureRandom) {
            console.log("✓ [SSLContext] 使用自定义 TrustManager");
            this.init(keyManager, null, secureRandom);
        };
    } catch (e) {
        console.log("! SSLContext hook 失败");
    }
}
```

```
        console.log(" [SSL Pinning Bypass] 配置完成\n");
    });
}
```

```
# 方式1：附加到运行中的 App
frida -U com.example.app -l bypass_ssl_pinning.js

# 方式2：启动 App 并注入
frida -U -f com.example.app -l bypass_ssl_pinning.js --no-pause
```

[SSL Pinning Bypass] 配置完成

方法 2: 使用 Xposed 模块

JustTrustMe 安装步骤

1. 确保设备已安装 Xposed Framework
2. 下载 JustTrustMe 模块: <https://github.com/Fuzion24/JustTrustMe>
3. 在 Xposed Installer 中激活
4. 重启设备

方法 3: 修改 APK (重打包)

APK 重打包步骤

如果 Frida 被检测，可以修改 APK 来信任用户证书：

1. 反编译 APK
2. 修改 `AndroidManifest.xml`，添加：

```
<application android:networkSecurityConfig="@xml/network_security_config">
```

1. 创建 `res/xml/network_security_config.xml`：

```
<network-security-config>
    <base-config cleartextTrafficPermitted="true">
        <trust-anchors>
            <certificates src="system" />
            <certificates src="user" />
        </trust-anchors>
    </base-config>
</network-security-config>
```

1. 重新打包并签名 APK

工作原理

MITM（中间人攻击）流程

1. App 发起 HTTPS 请求
2. 代理解密请求（使用安装的证书）
3. 代理重新加密并转发到真实服务器
4. 服务器响应经过代理返回给 App

SSL Pinning 是什么？

App 内置了服务器证书的指纹（Hash），只信任特定证书：

```
CertificatePinner pinner = new CertificatePinner.Builder()
    .add("api.example.com", "sha256/AAAAAAAAAA...")
    .build();
```

常见问题

问题 1: 手机无法连接代理

症状: 浏览器显示"无法连接到代理服务器"

检查:

1. PC 和手机是否在同一 Wi-Fi?
2. PC 防火墙是否允许 8888 端口?

```
# Windows 防火墙规则 (以管理员身份运行)
netsh advfirewall firewall add rule name="Burp Proxy" dir=in action=allow protocol=TCP
localport=8888

# macOS
# 系统偏好设置 → 安全性与隐私 → 防火墙选项 → 允许 Java
```

```
# 检查端口
netstat -an | grep 8888 # macOS/Linux
netstat -an | findstr 8888 # Windows
```

问题 2: 证书无效

症状: 浏览器显示证书无效

Android 7.0+ 限制:

- 默认只信任系统证书
- 必须将证书安装到 `/system/etc/security/cacerts/` (需要 Root)

无 Root 设备的解决方案:

- 使用 Magisk + MagiskTrustUserCerts 模块
- 或修改 APK (参考方法 3)

问题 3: Frida 脚本不生效

可能原因:

1. App 使用了自定义网络库 → 需要定位具体的类名和方法, 定制 Hook 脚本
2. Frida 被检测 → 使用重命名的 frida-server:

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

问题 4: 抓不到某些请求

可能原因:

1. 使用了 HTTP/2 或 QUIC → Burp Suite → Proxy → Options → HTTP/2 → 勾选"Enable HTTP/2"
2. 直接使用 Socket 通信 → 需要使用 Wireshark 或 tcpdump 抓取原始 TCP 包
3. 加密的自定义协议 → 需要逆向分析加密算法并解密

延伸阅读

相关 Recipes

- [密码学分析](#) - 分析 API 签名和加密算法
- [Frida 反调试绕过](#) - 如果 App 检测到 Frida
- [TLS 指纹分析](#) - 理解 TLS 指纹技术

工具深入

- [Frida 完整指南](#)

-
- [Burp Suite 使用技巧] - TODO, 一个比较流氓的工具

案例分析

- 音乐 App 分析 - API 抓包实战
 - 社交媒体风控 - 复杂签名分析
-

快速参考

一键启动脚本

- macOS/Linux:

```
#!/bin/bash
# start_proxy.sh

# 获取本机 IP
IP=$(ifconfig | grep "inet " | grep -v 127.0.0.1 | awk '{print $2}' | head -1)

echo "代理地址: $IP:8888"
echo "配置手机代理到: $IP:8888"
echo "证书下载: http://burp (Burp) 或 http://mitm.it (mitmproxy)"
echo ""

# 启动mitmproxy
mitmweb -p 8888 --listen-host 0.0.0.0
```

- Windows:

```
@echo off
REM start_proxy.bat

for /f "tokens=2 delims=:" %%a in ('ipconfig ^| findstr /c:"IPv4"') do set IP=%%a
echo 代理地址: %IP%:8888
echo 配置手机代理到: %IP%:8888
pause

java -jar burpsuite_community.jar
```

快速 SSL Pinning 绕过

```
# 下载通用 SSL Pinning 绕过脚本
curl -O https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/

# 运行
frida -U -f com.target.app -l universal-ssl-pinning.js --no-pause
```

[R02] 加密算法分析

R02: 分析并提取 Android 应用的加密密钥

问题场景

你遇到了什么问题？

- App 的 API 请求参数被加密了，看不懂内容
- 想知道 App 使用了什么加密算法
- 需要提取加密密钥来解密数据
- 想重现 App 的加密/签名逻辑用于自动化
- 需要绕过加密验证或签名检查

本配方教你：识别加密算法、定位密钥位置、使用 Frida 动态提取密钥。

核心理念：

💡 密码学逆向的关键不是破解算法，而是找到密钥

- 不要试图“破解”AES/RSA 等成熟算法（几乎不可能）
- 用动态分析直接提取密钥
- 或直接调用 App 自己的加密函数（利用已有密钥）

预计用时：30-60 分钟

工具清单

必需工具

- jadx-gui - Java 层静态分析
- Frida - 动态 Hook 提取密钥
- Android 设备 (已 Root)

可选工具

- IDA Pro / Ghidra - Native 层分析
- Burp Suite - 抓包查看加密后的数据
- CyberChef - 在线加密/解密工具 (<https://gchq.github.io/CyberChef/>)

前置条件

✓ 确认清单

```
# 1. Frida 正常运行  
frida-ps -U  
  
# 2. jadx-gui 已安装  
jadx-gui --version  
  
# 3. 抓包环境已配置 (可选)  
# 参考: network_sniffing.md
```

解决方案

第1步：识别加密算法（5分钟）

1.1 搜索特征字符串

用 jadx-gui 打开 APK，全局搜索：

```
# 对称加密  
AES  
DES  
3DES
```

```
# 非对称加密  
RSA  
ECC
```

```
# 哈希算法  
MD5  
SHA  
SHA256  
HMAC
```

```
# 加密模式  
ECB  
CBC  
CTR  
GCM
```

```
# Padding  
PKCS5Padding  
PKCS7Padding
```

```
# 编码  
Base64
```

1.2 搜索 Java 加密 API

```
// Java 层加密 API  
javax.crypto.Cipher  
javax.crypto.spec.SecretKeySpec  
javax.crypto.spec.IvParameterSpec  
javax.crypto.Mac  
java.security.Signature  
java.security.MessageDigest  
  
// Base64 编码  
android.util.Base64  
java.util.Base64
```

1.3 检查 Native 层加密

```
# 解压 APK  
unzip app.apk -d app_unzipped  
  
# 搜索 .so 文件中的加密库  
strings app_unzipped/lib/*lib*.so | grep -i -E "openssl|crypto|encrypt|aes|rsa"  
  
# 或使用 rabin2 分析  
rabin2 -z app_unzipped/lib/arm64-v8a/libnative.so | grep -i encrypt
```

第 2 步：定位加密代码（10 分钟）

2.1 跟踪加密字符串

假设你搜到了 `AES/CBC/PKCS5Padding`：

1. 在 jadx 中点击这个字符串
2. 查看交叉引用 (`X` 键或右键 → Find Usage)
3. 跳转到使用这个字符串的函数

典型代码模式：

```
// 你可能会看到类似这样的代码
public static String encrypt(String plaintext) {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    SecretKeySpec key = new SecretKeySpec(KEY_BYTES, "AES");
    IvParameterSpec iv = new IvParameterSpec(IV_BYTES);
    cipher.init(Cipher.ENCRYPT_MODE, key, iv);
    byte[] encrypted = cipher.doFinal(plaintext.getBytes());
    return Base64.encodeToString(encrypted, Base64.DEFAULT);
}
```

记录关键信息：

```
密钥变量: KEY_BYTES
IV 变量: IV_BYTES
加密函数: com.example.app.CryptoUtils.encrypt()
```

第 3 步：找到密钥位置（10 分钟）

📍 位置 1: Java 代码硬编码（难度：低）

查找方法：

```
// 搜索关键词
SecretKeySpec
byte[] key
private static final byte[]
```

示例：

```
private static final byte[] KEY = {
    0x12, 0x34, 0x56, 0x78,
    0x9a, 0xbc, 0xde, 0xf0,
    // ... 16/24/32 字节
};
```

密钥长度：

- AES-128: 16 字节

- AES-192: 24 字节
- AES-256: 32 字节

📍 位置 2: 资源文件 (难度: 低)

查找路径:

```
# 检查 assets 目录  
ls app_unzipped/assets/  
  
# 检查 res/raw  
ls app_unzipped/res/raw/  
  
# 搜索二进制文件  
find app_unzipped -type f -exec file {} \; | grep data
```

常见文件名:

- `secret.key`
- `config.dat`
- `license.bin`

📍 位置 3: Native (.so) 硬编码 (难度: 中)

IDA Pro 分析:

1. 打开 `.so` 文件
2. 跳转到 Strings 窗口 (`Shift+F12`)
3. 搜索关键字符串
4. 查看交叉引用找到使用密钥的函数

Ghidra 分析:

1. 导入 `.so` 文件
2. 搜索 → For Strings
3. 筛选长度为 16/24/32 的可疑字符串

📍 位置 4: 动态生成 (难度: 高)

特征：密钥通过算法计算，常见方式：

```
// 基于设备信息生成  
String deviceId = getDeviceId();  
byte[] key = MD5(deviceId + SALT);  
  
// 基于时间戳  
long timestamp = System.currentTimeMillis();  
byte[] key = HMACSHA256(timestamp, SECRET);
```

从服务器获取：

- 启动时从服务器获取密钥
- 可能经过 RSA 加密传输

对策：

1. 抓包查看密钥传输
2. Hook 网络请求获取密钥
3. 或直接 Hook 加密函数（密钥已在内存中）

第 4 步：动态提取密钥（15 分钟）

终极方法：无论密钥藏在哪，只要加密函数被调用，Hook 就能抓到

4.1 Hook Java 层加密

通用 AES Hook 脚本 `dump_aes_key.js`：

```
Java.perform(function () {
    console.log("\n[Crypto Hook] 启动\n");

    // Hook Cipher.init
    var Cipher = Java.use("javax.crypto.Cipher");
    Cipher.init.overload(
        "int",
        "java.security.Key",
        "java.security.spec.AlgorithmParameterSpec"
    ).implementation = function (opmode, key, spec) {
        console.log("\n🔒 [Cipher.init] 捕获!");

        // 模式
        var mode = opmode == 1 ? "ENCRYPT" : "DECRYPT";
        console.log("    模式: " + mode);

        // 算法
        console.log("    算法: " + this.getAlgorithm());

        // 提取密钥
        try {
            var secretKey = Java.cast(
                key,
                Java.use("javax.crypto.spec.SecretKeySpec")
            );
            var keyBytes = secretKey.getEncoded();
            var Base64 = Java.use("android.util.Base64");
            console.log("    密钥 (Base64): " + Base64.encodeToString(keyBytes, 0));
            console.log("    密钥 (Hex): " + bytesToHex(keyBytes));
        } catch (e) {
            console.log("    密钥类型: " + key.$className);
        }

        // 提取 IV
        if (spec) {
            try {
                var ivSpec = Java.cast(
                    spec,
                    Java.use("javax.crypto.spec.IvParameterSpec")
                );
                var ivBytes = ivSpec.getIV();
                console.log("    IV (Hex): " + bytesToHex(ivBytes));
            } catch (e) {}
        }
    }

    return this.init(opmode, key, spec);
};

// Hook Cipher.doFinal
Cipher.doFinal.overload("[B]").implementation = function (input) {
    var result = this.doFinal(input);
```

```
console.log("\n📦 [Cipher.doFinal] 捕获!");
console.log("    输入长度: " + input.length);
console.log("    输出长度: " + result.length);
console.log("    输入数据 (前32字节): " + bytesToHex(input.slice(0, 32)));
console.log("    输出数据 (前32字节): " + bytesToHex(result.slice(0, 32)));

return result;
};

function bytesToHex(bytes) {
    var hex = [];
    for (var i = 0; i < bytes.length && i < 32; i++) {
        hex.push(("0" + (bytes[i] & 0xff).toString(16)).slice(-2));
    }
    return hex.join(" ");
}

console.log("✅ [Crypto Hook] 配置完成\n");
});
```

运行脚本：

```
frida -U -f com.example.app -l dump_aes_key.js
```

预期输出：

```
🔒 [Cipher.init] 捕获!
模式: ENCRYPT
算法: AES/CBC/PKCS5Padding
密钥 (Base64): MTIzNDU2Nzg5MGFiY2RlZg==
密钥 (Hex): 31 32 33 34 35 36 37 38 39 30 61 62 63 64 65 66
IV (Hex): 66 65 64 63 62 61 30 39 38 37 36 35 34 33 32 31

📦 [Cipher.doFinal] 捕获!
输入长度: 128
输出长度: 144
输入数据 (前32字节): 7b 22 75 73 65 72 6e 61 6d 65 22 3a ...
输出数据 (前32字节): a3 b2 c1 d0 e4 f5 ...
```

4.2 Hook Native 层加密

查找 Native 加密函数：

```
# 使用 nm 查看函数  
nm -D libnative.so | grep -i encrypt  
  
# 使用 Frida  
frida -U -f com.example.app  
> Module.enumerateExports('libnative.so').filter(e => e.name.includes('encrypt'))
```

Hook Native 函数：

```
Interceptor.attach(  
    Module.findExportByName("libnative.so", "Java_com_example_Crypto_encrypt"),  
    {  
        onEnter: function (args) {  
            console.log("\n[Native Encrypt] 调用!");  
  
            // args[0] = JNIEnv*  
            // args[1] = jclass  
            // args[2] = 第一个参数 (通常是明文)  
            // args[3] = 第二个参数 (可能是密钥)  
  
            // 读取字符串参数  
            var plaintext = Java.vm  
                .getEnv()  
                .getStringUtfChars(args[2], null)  
                .readCString();  
            console.log("    明文: " + plaintext);  
  
            // 保存指针用于后续读取  
            this.keyPtr = args[3];  
        },  
        onLeave: function (retval) {  
            // retval 是返回值 (密文)  
            console.log("    返回值: " + retval);  
        },  
    }  
);
```

第 5 步：验证密钥（5 分钟）

5.1 使用 CyberChef 验证

1. 打开 <https://gchq.github.io/CyberChef/>

2. 选择操作： AES Decrypt

3. 输入：

```
| 项目 | 说明 |
| --- | --- |
| **Key** (Hex) | `31 32 33 34 ...` |
| **IV** (Hex) | `66 65 64 63 ...` |
| **Mode** | `CBC` |
| **Input** | 密文 (Base64 或 Hex) |
```

1. 点击 Bake!

如果解密成功，说明密钥正确！

5.2 Python 脚本验证

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import base64

# 从 Frida 获取的密钥和 IV (Hex 转 bytes)
key = bytes.fromhex('31323334353637383930616263646566')
iv = bytes.fromhex('66656463626130393837363534333231')

# 从抓包获取的密文
ciphertext = base64.b64decode('YWJjZGVmZ2hpamtsbW5vcA==')

# 解密
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)

print("解密结果:", plaintext.decode())
```

5.3 直接调用 App 的加密函数

```
Java.perform(function () {
    var CryptoUtils = Java.use("com.example.app.CryptoUtils");

    // 调用加密函数
    var encrypted = CryptoUtils.encrypt("Hello World");
    console.log("加密结果: " + encrypted);

    // 调用解密函数
    var decrypted = CryptoUtils.decrypt(encrypted);
    console.log("解密结果: " + decrypted);
});
```

工作原理

常见加密算法对照表

算法类型	算法	密钥长度	用途
对称加密	AES	128/192/256 bit	加密数据
	DES	56 bit	旧标准（不安全）
	3DES	168 bit	DES 增强版
非对称加密	RSA	1024/2048/4096 bit	密钥交换、签名
	ECC	256/384/521 bit	RSA 的高效替代
哈希	MD5	128 bit 输出	校验（不安全）
	SHA-256	256 bit 输出	安全哈希
	HMAC	可变	带密钥的哈希

AES 加密流程

```
明文数据
↓
[AES Encrypt] ← 使用密钥 + IV
↓
密文数据
↓
[Base64 Encode] ← 便于传输
↓
最终密文
```

常见问题

✗ 问题 1: Hook 脚本不生效

症状: 运行 Frida 脚本后没有任何输出

检查:

1. 确认加密函数被调用了吗?

```
Java.use("javax.crypto.Cipher").$init.overload().implementation = function () {
    console.log("[TEST] Cipher 实例化");
    return this.$init();
};
```

1. 加密是否在 Native 层?

- 改用 Native Hook

2. 类名可能被混淆

- 搜索所有包含 `Cipher` 的类:

```
Java.enumerateLoadedClasses({
    onMatch: function (className) {
        if (className.toLowerCase().includes("cipher")) {
            console.log(className);
        }
    },
    onComplete: function () {},
});
```

✖ 问题 2: 解密失败

可能原因:

1. IV 不正确

- 确认是否使用了 IV
- 某些实现会将 IV 附加在密文开头

2. Padding 不匹配

- 尝试不同的 Padding: `PKCS5Padding`, `PKCS7Padding`, `NoPadding`

3. 编码问题

```
# 尝试不同编码
ciphertext = base64.b64decode(data)      # Base64
ciphertext = bytes.fromhex(data)           # Hex
ciphertext = data.encode()                 # UTF-8
```

1. 密钥派生

- 可能使用了 PBKDF2 等密钥派生函数
- Hook `SecretKeyFactory.generateSecret()` 查看

✖ 问题 3: Native 函数找不到

症状: `Module.findExportByName()` 返回 `null`

解决：

1. 函数可能未导出

```
# 查看所有符号 (包括未导出)
readelf -s libnative.so | grep encrypt
```

1. 使用地址偏移 Hook

```
var baseAddr = Module.findBaseAddress('libnative.so');
var funcAddr = baseAddr.add(0x1234); // 从 IDA 获取偏移
Interceptor.attach(funcAddr, { ... });
```

1. 动态注册的 JNI 方法

```
// Hook RegisterNatives
var RegisterNatives = Module.findExportByName(
    "libart.so",
    "_ZN3art3JNI15RegisterNativesEP7_JNIEnvP7_jclassPK15JNINativeMethodi"
);
Interceptor.attach(RegisterNatives, {
    onEnter: function (args) {
        var methods = ptr(args[2]);
        console.log("Register JNI Method:", methods.readCString());
    },
});
```

✖ 问题 4: 密钥是字符串而非字节数组

症状：

```
// 看到类似这样的代码
SecretKeySpec key = new SecretKeySpec("MyPassword123".getBytes(), "AES");
```

解决方案：

1. 使用密钥派生函数 (KDF)

```
from Crypto.Protocol.KDF import PBKDF2

password = "MyPassword123"
salt = b"somesalt" # 需要从代码中找到
key = PBKDF2(password, salt, dkLen=16) # 16 字节 AES-128
```

说明:

- `PBKDF2` 会将任意长度的密码派生为固定长度的密钥
- `salt` 通常在代码中硬编码或从服务器获取
- `dkLen` 决定输出密钥长度: 16 (AES-128) / 24 (AES-192) / 32 (AES-256)

2. Hook 密钥派生函数

```
var SecretKeyFactory = Java.use("javax.crypto.SecretKeyFactory");
SecretKeyFactory.generateSecret.implementation = function (keySpec) {
    var key = this.generateSecret(keySpec);
    console.log("[密钥派生] 算法:", this.getAlgorithm());
    console.log("[密钥派生] 密钥 (Hex):", bytesToHex(key.getEncoded()));

    // 尝试获取 salt (如果是 PBEKeySpec)
    try {
        var PBEKeySpec = Java.use("javax.crypto.spec.PBEKeySpec");
        var pbeSpec = Java.cast(keySpec, PBEKeySpec);
        console.log("[密钥派生] Salt:", bytesToHex(pbeSpec.getSalt()));
        console.log("[密钥派生] 迭代次数:", pbeSpec.getIterationCount());
    } catch (e) {}

    return key;
};
```

3. 直接使用字符串密钥

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import hashlib

# 从代码中找到的字符串密码
password = "MyPassword123"

# 方法 1：直接使用前 16 字节
key = password.encode()[:16].ljust(16, b'\0')

# 方法 2：MD5 哈希（常见做法，输出正好 16 字节）
key = hashlib.md5(password.encode()).digest()

# 方法 3：SHA256 前 16 字节
key = hashlib.sha256(password.encode()).digest()[:16]

# 然后用于解密
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
```

延伸阅读

相关 Recipes

- [网络抓包](#) - 获取加密后的数据样本
- [Frida 反调试](#) - 如果 App 检测到 Hook
- [Native Hook 模式](#) - 深入 Native 层分析

工具深入

- [Frida 完整指南](#)
- [IDA Pro 使用](#)

案例分析

- [音乐 App 分析](#) - 加密音频格式分析
- [社交媒体风控](#) - API 签名算法逆向

理论基础

- 密码学基础知识 - TODO

快速参考

Hook 脚本模板库

1. RSA Hook

```
var Cipher = Java.use("javax.crypto.Cipher");
Cipher.init.overload("int", "java.security.Key").implementation = function (
    opmode,
    key
) {
    console.log("[RSA] 模式:", opmode == 1 ? "ENCRYPT" : "DECRYPT");
    console.log("[RSA] 密钥类型:", key.$className);

    // 获取公钥/私钥
    if (key.$className.includes("PublicKey")) {
        console.log("[RSA] 公钥:", bytesToHex(key.getEncoded()));
    } else if (key.$className.includes("PrivateKey")) {
        console.log("[RSA] 私钥:", bytesToHex(key.getEncoded()));
    }

    return this.init(opmode, key);
};
```

2. HMAC Hook

```
var Mac = Java.use("javax.crypto.Mac");
Mac.init.implementation = function (key) {
    console.log("[HMAC] 算法:", this.getAlgorithm());

    var secretKey = Java.cast(key, Java.use("javax.crypto.spec.SecretKeySpec"));
    console.log("[HMAC] 密钥:", bytesToHex(secretKey.getEncoded()));

    return this.init(key);
};

Mac.doFinal.overload("[B]").implementation = function (data) {
    var result = this.doFinal(data);
    console.log("[HMAC] 输入:", bytesToHex(data));
    console.log("[HMAC] 输出:", bytesToHex(result));
    return result;
};
```

3. Base64 Hook

```
var Base64 = Java.use("android.util.Base64");
Base64.decode.overload("java.lang.String", "int").implementation = function (
    str,
    flags
) {
    var result = this.decode(str, flags);
    console.log("[Base64] 解码:");
    console.log("    输入:", str.substring(0, 50) + "...");
    console.log("    输出 (Hex):", bytesToHex(result));
    return result;
};
```

OpenSSL 命令速查

```
# AES 加密
echo "Hello" | openssl enc -aes-128-cbc -K 31323334353637383930616263646566 -iv
66656463626130393837363534333231 -base64

# AES 解密
echo "密文" | base64 -d | openssl enc -d -aes-128-cbc -K
31323334353637383930616263646566 -iv 66656463626130393837363534333231

# 生成 MD5
echo -n "text" | openssl md5

# 生成 SHA256
echo -n "text" | openssl sha256

# RSA 密钥生成
openssl genrsa -out private.pem 2048
openssl rsa -in private.pem -pubout -out public.pem
```

[R03] TLS 指纹识别

R03: 使用 TLS 指纹识别检测和绕过应用指纹

问题场景

你遇到了什么问题？

- 你的自动化脚本被服务器识别并封禁了
- 你用 Python/curl 请求 API，但服务器返回 403/风控拦截
- 你想伪装成真实浏览器/官方 App 的 TLS 指纹
- 你想分析 App 使用的 TLS 库和配置
- 你想检测自己的请求是否暴露了异常的 TLS 特征

本配方教你：理解 TLS 指纹识别原理、如何检测自己的 TLS 指纹、以及如何伪造合法的 TLS 指纹。

核心理念：

💡 TLS 指纹是应用的“DNA”

- TLS 握手阶段暴露了客户端使用的库和配置
- 不同的 HTTP 客户端有不同的 TLS 指纹
- 服务器可以通过 JA3/JA4 指纹识别你的真实身份
- 即使使用 HTTPS，TLS 握手特征也是明文的

预计用时：20-40 分钟

工具清单

必需工具

- Wireshark - 抓取 TLS 握手包
- 在线 JA3 检测工具 - <https://ja3er.com> 或 <https://tls.peet.ws>
- Python 3.7+ - 用于脚本测试

可选工具

- curl-impersonate - 伪装浏览器 TLS 指纹的 curl
- tls-client (Python) - 支持自定义 TLS 指纹的 HTTP 库
- Burp Suite - 抓包分析
- ja3transport (Go) - Go 语言的 TLS 伪装库

前置条件

确认清单

1. Wireshark 已安装并可用
2. Python 3.7+ 环境配置完成

```
# 验证 Wireshark 安装  
wireshark --version  
  
# 验证 Python 环境  
python3 --version  
  
# 安装必要的 Python 库  
pip3 install requests pycurl tls-client
```

解决方案

第 1 步：理解 TLS 指纹识别原理（5 分钟）

1.1 什么是 JA3 指纹？

JA3 是一种通过分析 TLS `Client Hello` 包生成指纹的技术。

提取的字段：

1. TLS 版本（如 TLS 1.3 = 771）
2. 加密套件列表（Cipher Suites）
3. 扩展列表（Extensions）
4. 椭圆曲线列表（Elliptic Curves）
5. 椭圆曲线点格式（EC Point Formats）

生成过程：

```
拼接成String: "771,4865-4866-4867,0-23-65281,29-23-24,0"  
↓  
Calculate MD5 哈希  
↓  
JA3 指纹: e7d705a3286e19ea42f587b344ee6865
```

特性	MD5 指纹	JA3 指纹
格式	MD5 哈希	结构化字符串
可读性	无	高（包含版本、计数等）
示例	e7d705a3286e19ea42f587b344ee6865	t13d1516h2_174735a34e8a_b2149a751699
优势	简单，广泛支持	可模糊匹配，抗干扰

✓ 关键点：不同的 HTTP 库有不同的 JA3 指纹

客户端	JA3 指纹
Chrome 120	579cce312d18482fc42e2b822ca2430
Firefox 121	3b5074b1b5d032e5620f69f9f700ff0e
Python requests	084c44f52a434da89e0b1bc98f8dd159
curl 默认	51c64c77e60f3980eea90869b68c58a8

问题：如果你用 Python requests 访问服务器，即使设置了 User-Agent，服务器也能通过 JA3 识别出你不是真实浏览器

第 2 步：检测你的 TLS 指纹 (10 分钟)

2.1 在线检测

方法 1：访问 JA3 检测网站

```
# 用 curl 测试
curl https://ja3er.com/json

# 用 Python requests 测试
python3 << EOF
import requests
r = requests.get('https://ja3er.com/json')
print(r.text)
EOF
```

```
{
    "ja3": "084c44f52a434da89e0b1bc98f8dd159",
    "ja3_text":
    "771,4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53,0-23
-65281-10-11-35-16-5-13-18-51-45-43-27-21,29-23-24,0",
    "User-Agent": "python-requests/2.31.0"
}
```

```
curl -s https://tls.peet.ws/api/all | jq .
```

1. 打开 Wireshark
2. 过滤器输入：`tls.handshake.type == 1` (只显示 Client Hello)
3. 在终端执行请求：

```
curl https://example.com
```

接着展开 Transport Layer Security → Handshake Protocol: Client Hello

查看关键字段：

```
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
- ...
Extension: supported_groups (len=10)
- secp256r1 (0x0017)
- x25519 (0x001d)
- ...
```

✓ 成功标志：你已经看到了自己客户端的 TLS 握手特征

第 3 步：获取目标指纹（5 分钟）

目标：获取真实浏览器或官方 App 的 JA3 指纹用于伪装

3.1 浏览器指纹

方法 1：直接查询

Chrome 浏览器访问 <https://ja3er.com/json>

记录显示的 JA3 哈希值。

方法 2：从 GitHub 数据库查询

访问 <https://github.com/salesforce/ja3/blob/master/lists/osx-nix-jas.csv>

搜索 "Chrome"、"Safari"、"Firefox" 找到对应版本的 JA3。

3.2 android App 指纹

使用 Wireshark 抓取真实 App 的流量：

1. 配置手机走电脑代理
2. Wireshark 监听对应网卡
3. 打开目标 App，触发网络请求
4. 过滤 `tls.handshake.type == 1` 找到 Client Hello 包
5. 记录或导出该包

提取 JA3:

```
# Use ja3 Tool (NeedInstall)
pip3 install pyshark
python3 << 'EOF'
import pyshark
cap = pyshark.FileCapture('capture.pcap', display_filter='tls.handshake.type == 1')
for pkt in cap:
    print(pkt.tls.handshake_ciphersuite)
EOF
```

第 4 步：伪造 TLS 指纹（15 分钟）

4.1 使用 curl-impersonate（推荐）

curl-impersonate 是一个修改版的 curl，能完美模拟浏览器的 TLS 指纹。

安装（macOS）：

```
# Use Homebrew
brew install curl-impersonate

# or download pre-compiled version
# https://github.com/lwthiker/curl-impersonate/releases
```

```
# 伪装成 Chrome 120
curl_chrome120 https://ja3er.com/json

# 伪装成 Firefox 121
curl_ff121 https://ja3er.com/json

# 伪装成 Safari 17
curl_safari17 https://ja3er.com/json
```

4.2 使用 Python tls-client 库

安装：

```
pip3 install tls-client
```

基本使用:

```
import tls_client

# 创建会话, 伪装成 Chrome 120
session = tls_client.Session(
    client_identifier="chrome_120",
    random_tls_extension_order=True
)

# 发送请求
response = session.get("https://ja3er.com/json")
print(response.json())
```

支持的客户端标识符:

```
# 桌面浏览器
"chrome_103", "chrome_110", "chrome_120"
"firefox_102", "firefox_104", "firefox_121"
"safari_15_3", "safari_16_0", "safari_17_0"

# 移动端
"okhttp4_android_7", "okhttp4_android_8", "okhttp4_android_13"
```

自定义 JA3 字符串:

```
import tls_client

session = tls_client.Session(
    client_identifier="custom",
    ja3_string="771,4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157
    -47-53,0-23-65281-10-11-35-16-5-13-18-51-45-43-27-21,29-23-24,0"
)
```

4.3 使用 Go ja3transport 库

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "github.com/CUCyber/ja3transport"
)

func main() {
    // 创建带 JA3 指纹的 Transport
    tr, _ := ja3transport.NewTransport("771,4865-4866-4867-49195-49199-49196-49200-52393-52392-4917
    1-49172-156-157-47-53,0-23-65281-10-11-35-16-5-13-18-51-45-43-27-21,29-23-24,0")

    client := &http.Client{Transport: tr}

    resp, _ := client.Get("https://ja3er.com/json")
    defer resp.Body.Close()

    body, _ := io.ReadAll(resp.Body)
    fmt.Println(string(body))
}
```

第 5 步：验证伪装效果（5 分钟）

创建对比脚本 `compare_ja3.sh`：

```
#!/bin/bash

echo "==== 原生 curl ==="
curl -s https://ja3er.com/json | jq -r '.ja3'

echo ""
echo "==== curl-impersonate (Chrome) ==="
curl_chrome120 -s https://ja3er.com/json | jq -r '.ja3'

echo ""
echo "==== Python requests ==="
python3 -c "import requests; print(requests.get('https://ja3er.com/json').json()['ja3'])"

echo ""
echo "==== Python tls-client ==="
python3 << EOF
import tls_client
session = tls_client.Session(client_identifier="chrome_120")
print(session.get("https://ja3er.com/json").json()['ja3'])
EOF
```

运行对比：

```
chmod +x compare_ja3.sh
./compare_ja3.sh
```

预期输出：

```
==== 原生 curl ===
51c64c77e60f3980eea90869b68c58a8

==== curl-impersonate (Chrome) ===
579cce312d18482fc42e2b822ca2430

==== Python requests ===
084c44f52a434da89e0b1bc98f8dd159

==== Python tls-client ===
579cce312d18482fc42e2b822ca2430
```

实战示例 - 访问受保护的 API：

```
import tls_client

# 使用伪装的 TLS 指纹
session = tls_client.Session(client_identifier="chrome_120")

headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36'
}

response = session.get('https://api.example.com/protected', headers=headers)
print(response.status_code)
print(response.text)
```

伪装效果对比：

工具	User-Agent	TLS 指纹	服务器响应
requests	Chrome	Python	✗ 403 Forbidden
tls-client	Chrome	Chrome	✓ 200 OK

工作原理

TLS 握手过程



JA3 指纹生成细节

原始字符串示例：

JA4 指纹格式:

```
t13d1516h2_174735a34e8a_b2149a751699
|   |   |   |
|   |   |   |   |
|   |   |   |   |   Signature Algorithms 哈希
|   |   |   |   |   |
|   |   |   |   |   |   Extensions 哈希 (截断)
|   |   |   |   |   |   h2 = HTTP/2
|   |   |   |   |   |   16 个 Extensions
|   |   |   |   |   15 个 Cipher Suites
|   |   |   |   TLS 1.3
```

- 可读性强（无需查表）

常见问题

✖ 问题 1: curl-impersonate 安装失败

症状: Homebrew 找不到 curl-impersonate

解决：

```
# macOS/Linux: 手动下载预编译版本  
wget https://github.com/lwthiker/curl-impersonate/releases/download/v0.6.1/curl-  
impersonate-v0.6.1.x86_64-linux-gnu.tar.gz  
  
tar -xzf curl-impersonate-*.tar.gz  
cd curl-impersonate-*  
sudo cp curl_* /usr/local/bin/
```

✖ 问题 2: tls-client 不支持某个浏览器版本

症状: `ValueError: Unknown client identifier: chrome_999`

解决: 查看支持的客户端列表

```
import tls_client  
print(tls_client.settings.ClientIdentifiers)
```

✖ 问题 3: 伪装后仍被检测

可能原因:

1. HTTP/2 指纹不匹配

- JA3 指纹是浏览器, 但 HTTP 头顺序/值不对
 - 解决: 使用完整的浏览器模拟 (包括 HTTP/2 特征)

2. 行为特征异常

- 请求速度太快
- 缺少 Referer/Cookie
- 解决: 添加延迟、模拟真实用户行为

3. IP 信誉问题

- IP 被标记为数据中心/代理
 - 解决: 使用住宅代理或轮换 IP

4. 设备指纹

- 服务器检测 Canvas 指纹、WebGL 指纹等
- 解决：使用真实浏览器自动化（Selenium + undetected-chromedriver）

✖ 问题 4：如何在 Frida 中修改 TLS 指纹？

场景：你想修改 Android App 的 TLS 指纹

方法：Hook Java 层 SSLSocket

```
Java.perform(function () {
    var SSLSocket = Java.use("javax.net.ssl.SSLSocket");

    SSLSocket.setEnabledCipherSuites.implementation = function (suites) {
        console.log("[*] Original Cipher Suites:", suites);

        // 修改为目标指纹加密套件
        var customSuites = [
            "TLS_AES_128_GCM_SHA256",
            "TLS_AES_256_GCM_SHA384",
            "TLS_CHACHA20_POLY1305_SHA256",
        ];

        console.log("[*] ModifyAfter:", customSuites);
        return this.setEnabledCipherSuites(customSuites);
    };
});
```

延伸阅读

相关 Recipes

- [网络抓包](#) - 抓取 TLS 握手包
- [密码学分析](#) - 分析加密实现
- [JA3 指纹详解](#) - JA3 技术深入
- [JA4 指纹详解](#) - JA4+ 套件详解

工具深入

- curl-impersonate 文档 - <https://github.com/lwthiker/curl-impersonate>
- tls-client (Python) - <https://github.com/FlorianREGAZ/Python-Tls-Client>
- ja3transport (Go) - <https://github.com/CUCyber/ja3transport>

在线资源

- JA3 检测 - <https://ja3er.com>
- TLS 指纹检测 - <https://tls.peet.ws>
- JA3 数据库 - <https://github.com/salesforce/ja3>

理论基础

- [TLS 协议详解](#) - TODO
- [HTTP/2 指纹](#) - TODO

快速参考

常用工具对比

工具	语言	难度	特点
curl-impersonate	Bash		最简单, 完美模拟
tls-client	Python		易用, 支持多种浏览器
ja3transport	Go		高性能, 需要 Go 环境
requests + urllib3	Python		复杂, 需深度定制

快速检测脚本

detect_ja3.sh:

```
#!/bin/bash

echo "正在检测 TLS 指纹..."
echo ""

URL="https://ja3er.com/json"

# 检测当前客户端
JA3=$(curl -s "$URL" | jq -r '.ja3')
echo "你的 JA3: $JA3"

# 查询已知指纹
echo ""
echo "常见客户端 JA3:"
echo "  Chrome 120: 579cccf312d18482fc42e2b822ca2430"
echo "  Firefox 121: 3b5074b1b5d032e5620f69f9f700ff0e"
echo "  Safari 17: 4e2d5f6c3e8f7a9b0c1d2e3f4a5b6c7d"
echo "  Python req: 084c44f52a434da89e0b1bc98f8dd159"
echo "  curl:         51c64c77e60f3980eea90869b68c58a8"

# 对比
if [ "$JA3" == "579cccf312d18482fc42e2b822ca2430" ]; then
    echo ""
    echo "✅ 匹配: Chrome 120"
elif [ "$JA3" == "084c44f52a434da89e0b1bc98f8dd159" ]; then
    echo ""
    echo "⚠️ 匹配: Python requests (容易被识别)"
else
    echo ""
    echo "❓ 未知指纹"
fi
```

TLS 指纹伪装模板

browser_session.py:

```
"""
TLS 指纹伪装模板
"""

import tls_client

class BrowserSession:
    """模拟浏览器会话"""

    PROFILES = {
        'chrome': 'chrome_120',
        'firefox': 'firefox_121',
        'safari': 'safari_17_0',
        'android': 'okhttp4_android_13'
    }

    def __init__(self, browser='chrome'):
        """初始化会话

        Args:
            browser: 浏览器类型 ('chrome', 'firefox', 'safari', 'android')
        """

        identifier = self.PROFILES.get(browser, 'chrome_120')
        self.session = tls_client.Session(
            client_identifier=identifier,
            random_tls_extension_order=True
        )
        self._set_headers(browser)

    def _set_headers(self, browser):
        """设置对应 HTTP 头"""

        user_agents = {
            'chrome': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
            'firefox': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:121.0) Gecko/20100101 Firefox/121.0',
            'safari': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.0 Safari/605.1.15',
            'android': 'Mozilla/5.0 (Linux; Android 13) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.6099.144 Mobile Safari/537.36'
        }

        self.session.headers.update({
            'User-Agent': user_agents.get(browser, user_agents['chrome']),
            'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8',
            'Accept-Language': 'en-US,en;q=0.5',
            'Accept-Encoding': 'gzip, deflate, br',
            'DNT': '1',
            'Connection': 'keep-alive',
            'Upgrade-Insecure-Requests': '1'
        })


```

```
def get(self, url, **kwargs):
    """发送 GET 请求"""
    return self.session.get(url, **kwargs)

def post(self, url, **kwargs):
    """发送 POST 请求"""
    return self.session.post(url, **kwargs)

def verify_fingerprint(self):
    """验证 TLS 指纹"""
    r = self.get('https://ja3er.com/json')
    return r.json()

# 使用示例
if __name__ == '__main__':
    # 创建 Chrome 会话
    browser = BrowserSession('chrome')

    # 验证指纹
    print("验证 TLS 指纹...")
    result = browser.verify_fingerprint()
    print(f"JA3: {result['ja3']}")
    print(f"User-Agent: {result['User-Agent']}")

    # 发送请求
    response = browser.get('https://api.example.com/data')
    print(f"\n状态码: {response.status_code}")
```

[R04] JA3 指纹技术

R04: JA3 TLS 指纹识别技术详解

JA3 是一种创建 SSL/TLS 客户端指纹的方法，旨在轻松识别网络上的客户端应用程序。当客户端与服务器建立加密连接时，它首先会发送一个 `Client Hello` 包。这个包的格式和内容在很大程度上取决于用于创建连接的客户端应用程序（例如浏览器、恶意软件、移动 App）中的库和方法。JA3 通过收集 `Client Hello` 包中特定字段的值，并将它们组合成一个易于共享和比较的 MD5 哈希值，从而为客户端生成一个独特的"指纹"。

目录

- JA3 TLS 指纹识别技术详解
 - 目录
 - 工作原理
 - 指纹生成过程
 - JA3S - 服务器端指纹
 - 应用场景
 - 局限性
 - 如何检测 JA3

工作原理

JA3 指纹的核心思想是：客户端的 `Client Hello` 包暴露了其身份。

一个 `Client Hello` 包包含了客户端希望如何与服务器进行通信的各种细节。JA3 方法精确地选择了以下 5 个字段，并按照特定顺序将它们串联起来：

1. SSL/TLS Version (版本号): 客户端支持的最高 TLS 版本。

-
2. Accepted Ciphers (加密套件): 客户端愿意接受的加密套件列表，按其偏好顺序排列。
 3. List of Extensions (扩展列表): `Client Hello` 中包含的所有扩展，按其出现顺序排列。
 4. Elliptic Curves (椭圆曲线): 客户端支持的椭圆曲线列表。
 5. Elliptic Curve Point Formats (椭圆曲线点格式): 支持的点格式列表。

这些字段的组合对于特定的客户端应用程序（及其版本）来说通常是独一无二的。例如，Chrome 浏览器、Firefox 浏览器、Tor 浏览器和一个 Golang 编写的僵尸网络程序，它们生成的 `Client Hello` 在这些字段上会有明显的差异。

指纹生成过程

生成 JA3 指纹的步骤如下：

1. 收集字段值: 从一个 TCP 会话的 `Client Hello` 包中，提取上述 5 个字段的十进制值。
 2. 格式化和拼接:
 - 每个字段内的值用 `-` 分隔。
 - 5 个主要字段之间用 `,` 分隔。
 - 例如，一个 JA3 字符串看起来像这样:
- 771,4865-4866-4867-49195-49199-...,23-65281-10-11-35-16-0-13-18,29-23-24,0
3. 计算 MD5 哈希: 对上述拼接好的字符串计算 MD5 哈希值。
 4. 最终指纹: 得到的 32 位十六进制字符串就是该客户端的 JA3 指纹。
 - 例如，上述字符串的 MD5 哈希可能是: `e7d705a3286e19ea42f587b344ee6865`。

这个最终的 MD5 哈希就是可用于识别、共享和查询的 JA3 指纹。

JA3S - 服务器端指纹

与 JA3 对应, JA3S 是对服务器响应的指纹。它基于服务器在 `Server Hello` 包中选择的参数。JA3S 收集以下字段:

1. SSL/TLS Version
2. Selected Cipher
3. List of Extensions

将这些值拼接并进行 MD5 哈希, 就得到了 JA3S 指纹。

为什么 JA3S 很重要?

将 JA3 和 JA3S 结合起来, 可以提供对加密连接的更强洞察力。例如, 一个恶意软件 (JA3) 可能会尝试连接多个不同的 C2 服务器 (不同的 JA3S)。反之, 一个 C2 服务器 (JA3S) 可能会接受来自不同类型恶意软件 (不同的 JA3) 的连接。这种组合分析可以更精确地描绘出威胁活动的全貌。

应用场景

- **恶意软件家族识别:** 许多恶意软件家族 (如 Trickbot, Emotet) 使用特定的 SSL/TLS 库, 导致它们具有独特且一致的 JA3 指纹。安全分析师可以创建规则来检测或阻止已知的恶意 JA3 哈希。
 - **僵尸网络检测:** 僵尸网络中的客户端通常是相同的程序, 因此它们的 JA3 指纹也是相同的。这使得大规模识别受感染主机成为可能。
 - **威胁情报共享:** JA3 指纹是一个优秀的技术性"失陷指标"(IOC)。安全社区可以共享已知的恶意 JA3 列表, 就像共享恶意 IP 地址或域名一样。
 - **识别非标准应用:** 可以用于识别组织内部网络中不合规或非标准的应用程序。
-

局限性

尽管 JA3 非常有用，但它也有一些明显的缺点：

- 指纹冲突: 不同的应用程序可能偶然会使用相同的加密库和配置，从而产生相同的 JA3 指纹。
 - 容易被规避 (Spoofing): 只要攻击者有能力修改其客户端的 SSL/TLS 库，他们就可以刻意模仿一个常见、合法的应用程序（如 Chrome 浏览器）的 `Client Hello` 包，从而生成一个“合法”的 JA3 指纹来逃避检测。这种技术被称为“JA3 欺骗”。
 - 指纹随版本变化: 当一个合法应用（如 Chrome）更新时，它的 TLS 实现可能会改变，导致其 JA3 指纹也发生变化。这意味着维护一个准确的指纹数据库需要持续的努力。
 - 信息有限: 一个 MD5 哈希本身不包含任何信息。你无法从两个不同的哈希值看出它们对应的客户端有多相似。例如，Chrome 90 和 Chrome 91 的 JA3 哈希可能完全不同，即使它们的 `Client Hello` 包只有微小的差异。
-

如何检测 JA3

要实现 JA3 检测，你需要能够监控网络流量并解析 TLS 握手的工具。常见的实现方式包括：

- 网络安全监控 (NSM) 工具: Zeek (原名 Bro) 是原生支持 JA3 和 JA3S 指纹生成的黄金标准。
 - Suricata: 从 4.1 版本开始，Suricata 也内置了 JA3 指纹功能。
 - Wireshark: 可以通过特定的插件或手动的 tshark 脚本来提取和计算 JA3。
-

[R05] JA4 指纹技术

R05: JA4+ TLS/QUIC 指纹识别技术详解

JA4+ 是由 FoxIO (原 Salesforce 的 JA3 团队) 开发的一套网络指纹识别方法的集合，旨在成为 JA3 的下一代演进版本。它不仅仅是对 JA3 的简单升级，而是一个更全面、更具结构化和可操作性的指纹套件，旨在解决 JA3 的核心痛点，并扩展到 QUIC 和 HTTP 等协议。

目录

- JA4+ TLS/QUIC 指纹识别技术详解
 - 目录
 - 为什么需要 JA4+ (JA3 的局限性)
 - JA4 的核心设计 - 不再是哈希
 - JA4+ 套件概览
 - JA4 (客户端 TLS)
 - JA4S (服务器端 TLS)
 - JA4H (HTTP 客户端)
 - JA4X (TLS 证书)
 - JA4L (实验性)
 - JA4 vs JA3: 核心优势
 - 应用与实践

为什么需要 JA4+ (JA3 的局限性)

JA3 是一个非常成功的技术，但其核心设计——一个单一的 MD5 哈希——带来了几个无法克服的挑战：

- 缺乏上下文：一个 MD5 哈希是不透明的。`e7d705a3...` 和 `a8d9b1c2...` 这两个哈希值，我们无法判断它们代表的客户端有多相似。可能只是 TLS 扩展顺序的一个微小变化，就导致了完全不同的哈希。
- "雪崩效应"：客户端的任何微小更新（例如，Chrome 101 → 102）都可能导致 JA3 哈希完全改变，使得基于精确匹配的规则变得非常脆弱。
- 难以进行模糊搜索：无法进行"搜索所有使用 TLS 1.3 但不包含某个特定加密套件的客户端"这样的灵活查询。
- 易于被模仿：攻击者只需要精确复现 `Client Hello` 的特定字段，就能完全复制一个合法应用的 JA3 哈希。

JA4+ 的诞生就是为了解决这些问题。

JA4 的核心设计 - 不再是哈希

JA4 最大的革新是放弃了单一、不透明的哈希值，转而采用一种结构化、人类可读的字符串格式。这使得指纹本身就携带了丰富的上下文信息。

JA4 的指纹格式为：`Protocol_Version_Ciphers_Extensions_Signature`，每个部分都有特定的含义和构造方法。

一个典型的 JA4 指纹例子：`t13d1516h2_174735a34e8a_b2149a751699`

我们来分解它：

- `t` (Protocol)：协议。`t` 代表 TLS，`q` 代表 QUIC。
- `13` (TLS Version)：`Client Hello` 中支持的最高 TLS 版本。`12` = TLS 1.2，`13` = TLS 1.3。

- **d1516h2** (Ciphers & Extensions Count):
 - **d**: 客户端支持的加密套件是有序的 (sorted)。**i** 表示无序 (insipid)。
 - **15**: 客户端提供了 15 个加密套件。
 - **16**: 客户端提供了 16 个扩展。
 - **h2**: 客户端在 `Client Hello` 中使用了 2 个 GREASE (Generate Random Extensions And Sustain Extensibility) 值, 这通常是现代浏览器的特征。
- **_** (分隔符)
- **174735a34e8a** (Extensions): 这是对有序的扩展列表进行特定算法计算后得到的部分哈希。相似的扩展列表会产生相似的哈希前缀。
- **_** (分隔符)
- **b2149a751699** (Signature Algorithms): 这是对签名算法和支持的组 (椭圆曲线) 进行部分哈希计算后得到的值。

这种结构使得指纹既能用于精确匹配, 也能用于强大的模糊匹配。

JA4+ 套件概览

JA4+ 不是单一的工具, 而是一个方法论集合。

JA4 (客户端 TLS)

- 目标: 识别发起 TLS 连接的客户端应用。
- 格式: 如上所述的 `p_v_c_e_s` 结构。

JA4S (服务器端 TLS)

- 目标: 识别响应 TLS 连接的服务器应用。
- 格式: `p_v_c_e`, 比客户端指纹稍简单。
 - 例如: `t13d03_a06f30d07525`

-
- `t` = TLS, `13` = TLS 1.3, `d` = 有序, `03` = 3 个扩展, `a06...` = 扩展的部分哈希。
 - 应用: 将 JA4 和 JA4S 结合, 可以进行更精准的匹配, 例如"只告警这个特定 JA4 连接到这个特定 JA4S 的行为"。

JA4H (HTTP 客户端)

- 目标: 对 HTTP 请求进行指纹识别, 作为对 JA4 的补充。
- 格式: `p_m_v_h`
 - `p`: 协议 (`h`=HTTP/1, `h2`=HTTP/2)。
 - `m`: 请求方法 (`g`=GET, `p`=POST)。
 - `v`: HTTP 版本。
 - `h`: 对 HTTP Header 的特定组合进行哈希。
- 应用: 可以用来检测 JA4 欺骗。例如, 一个声称自己是 Chrome 的 JA4 指纹, 却发送了不符合 Chrome 行为的 JA4H 指纹, 这很可能是一个恶意客户端。

JA4X (TLS 证书)

- 目标: 对 TLS 证书链进行指纹识别。
- 应用: 快速识别自签名证书、特定恶意软件使用的证书等。

JA4L (实验性)

- L for Lightweight。这是一个更简单的版本, 只包含数字和计数, 不包含哈希。
 - 应用: 适用于性能极高或资源受限的环境, 提供基本的模糊匹配能力。
-

JA4 vs JA3：核心优势

特性	JA4+	JA3
格式	结构化字符串	单一 MD5 哈希
可读性	高，指纹本身包含信息	无
模糊匹配	原生支持，可按部分查询	否
上下文	丰富 (协议, 版本, 计数)	无
欺骗难度	更高，需匹配行为逻辑	较低，只需匹配字段
覆盖范围	TLS, QUIC, HTTP, Certificates	仅 TLS
健壮性	高，微小变化不影响大局	低，"雪崩效应"

应用与实践

JA4+ 的应用场景比 JA3 更广泛和深入：

- 高级威胁狩猎:

反检测

[R06] Frida 反调试绕过

R06: 绕过 App 对 Frida 的检测

问题场景

你遇到了什么问题？

- 运行 Frida 后 App 立即崩溃或闪退
- App 显示"检测到调试工具"并拒绝运行
- Hook 脚本加载后 App 无响应或进入安全模式
- 某些功能在 Frida 环境下被禁用
- App 频繁弹窗提示"运行环境异常"

本配方教你：识别 Frida 检测技术、使用 Hook 绕过检测、定制 Frida 避免特征。

核心理念：

用 Frida 对抗检测 Frida - 以子之矛攻子之盾

- 在 App 检测之前就 Hook 检测函数
- 修改检测结果让它"看不见" Frida
- 或干脆隐藏 Frida 的所有特征

预计用时：15-45 分钟（取决于检测复杂度）

工具清单

必需工具

- Frida - 动态插桩框架

-
- Android 设备 (已 Root)
 - 文本编辑器 - 编写绕过脚本

可选工具

- jadx-gui - 静态分析检测代码
 - IDA Pro / Ghidra - Native 层检测分析
 - 定制版 Frida - 终极解决方案
-

前置条件

确认清单

```
# 1. Frida 正常运行  
frida-ps -U  
  
# 2. 能正常 attach (无检测时)  
frida -U -f com.example.app  
  
# 3. Root 权限可用  
adb shell su
```

解决方案

第 1 步：识别检测类型 (5 分钟)

1.1 触发检测

运行 Frida 并观察现象：

```
# 使用 spawn 模式启动 App  
frida -U -f com.example.app --no-pause
```

```
# 观察输出和 App 行为
```

检测现象对照表：

现象	可能的检测方式
立即闪退	端口扫描、进程名检测
弹窗"检测到 Root/调试"	模块名检测、线程名检测
特定功能被禁用	Inline Hook 检测
随机崩溃/卡顿	多重检测组合

1.2 静态分析检测代码（可选）

用 jadx 搜索关键词：

```
# Frida 特征  
frida  
gum-js  
27042  
  
# 检测相关  
/proc/self/maps  
/proc/* cmdline  
pthread_create  
connect  
socket
```

典型检测代码示例：

```
public static boolean isFridaDetected() {  
    // 检查端口  
    if (checkPort(27042)) return true;  
  
    // 检查进程  
    if (findProcess("frida-server")) return true;  
  
    // 检查模块  
    if (checkMaps("frida-agent")) return true;  
  
    return false;  
}
```

第 2 步：基础绕过（10 分钟）

重命名 frida-server：

```
# 下载 frida-server  
# 重命名为无害名字  
mv frida-server-16.1.4-android-arm64 system_daemon  
  
# 推送到设备  
adb push system_daemon /data/local/tmp/  
adb shell "chmod 755 /data/local/tmp/system_daemon"  
  
# 使用非标准端口启动  
adb shell "/data/local/tmp/system_daemon -l 0.0.0.0:8888 &"
```

使用 spawn 模式（重要）：

```
# 推荐: Spawn 模式 (最早注入)  
frida -U -f com.example.app -l bypass.js --no-pause  
  
# 不推荐: Attach 模式 (检测代码可能已运行)  
frida -U com.example.app -l bypass.js
```

第 3 步：通用绕过脚本

```
Java.perform(function () {
    console.log("\n[Frida Anti-Detection] 已启动\n");

    // =====
    // 1. 绕过端口扫描检测
    // =====
    var connect = Module.findExportByName("libc.so", "connect");
    if (connect) {
        Interceptor.attach(connect, {
            onEnter: function (args) {
                var sockaddr = ptr(args[1]);
                var family = sockaddr.readU16();

                if (family === 2) {
                    // AF_INET
                    var port = (sockaddr.add(2).readU8() << 8) | sockaddr.add(3).readU8();
                    var ip = sockaddr.add(4).readU32();

                    // 检测是否在扫描 Frida 默认端口
                    if (port === 27042 || port === 27043) {
                        console.log("[Port] 拦截端口扫描: " + port);
                        // 修改端口为无效端口
                        sockaddr.add(2).writeU8(0xff);
                        sockaddr.add(3).writeU8(0xff);
                    }
                }
            },
        });
        console.log("[Port] Hook connect() 完成");
    }

    // =====
    // 2. 绕过 /proc/self/maps 检测
    // =====
    var fgets = Module.findExportByName("libc.so", "fgets");

    if (fgets) {
        Interceptor.attach(fgets, {
            onEnter: function (args) {
                this.buffer = args[0];
                this.fp = args[2];
            },
            onLeave: function (retval) {
                if (retval.isNull()) return;

                var line = this.buffer.readCString();
                if (line) {
                    // 隐藏 Frida 相关模块
                    if (
                        line.includes("frida") ||
                        line.includes("gum-js") ||
                        line.includes("frida-agent")
                    )
                }
            }
        });
    }
})
```

```
        )
        console.log("[Maps] 隐藏模块: " + line.substring(0, 50) + "...");
        // 替换为空行
        this.buffer.writeUtf8String("\n");
    }
}

},
});

console.log("[Maps] Hook fgets() 完成");
}

// =====
// 3. 绕过 strstr 字符串检测
// =====
var strstr = Module.findExportByName("libc.so", "strstr");
if (strstr) {
    Interceptor.attach(strstr, {
        onEnter: function (args) {
            this.haystack = args[0].readCString();
            this.needle = args[1].readCString();

            if (
                this.needle &&
                (this.needle.includes("frida") ||
                 this.needle.includes("gum-js") ||
                 this.needle === "frida-agent" ||
                 this.needle === "frida-server")
            ) {
                this.shouldBypass = true;
            }
        },
        onLeave: function (retval) {
            if (this.shouldBypass) {
                console.log("[Strstr] 隐藏字符串: " + this.needle);
                retval.replace(ptr(0)); // 返回NULL (未找到)
            }
        },
    });
    console.log("[Strstr] Hook strstr() 完成");
}

// =====
// 4. 绕过 Java 层检测函数
// =====
setTimeout(function () {
    // 搜索常见检测函数名
    var detectNames = [
        "isFridaDetected",
        "checkFrida",
        "detectDebugger",
        "isHooked",
        "checkRoot",
    ];
});
```

```
Java.enumerateLoadedClasses({
    onMatch: function (className) {
        try {
            var clazz = Java.use(className);
            detectNames.forEach(function (methodName) {
                if (clazz[methodName]) {
                    console.log(
                        "[Java] 找到检测函数: " + className + "." + methodName
                    );
                    clazz[methodName].implementation = function () {
                        console.log("[Java] 拦截调用: " + className + "." + methodName);
                        return false; // 返回"未检测到"
                    };
                }
            });
        } catch (e) {}
    },
    onComplete: function () {
        console.log("[Java] 类枚举完成");
    },
});
}, 500);

// =====
// 5. 绕过线程名检测
// =====
var pthread_setname_np = Module.findExportByName(
    "libc.so",
    "pthread_setname_np"
);
if (pthread_setname_np) {
    Interceptor.attach(pthread_setname_np, {
        onEnter: function (args) {
            var threadName = args[1].readCString();
            if (
                threadName &&
                (threadName.includes("gum-js") ||
                 threadName.includes("gmain") ||
                 threadName.includes("pool-"))
            ) {
                console.log("[Thread] 修改线程名: " + threadName + " → normal");
                args[1].writeUtf8String("normal");
            }
        },
    });
    console.log("[Thread] Hook pthread_setname_np() 完成");
}

console.log("\n[Frida Anti-Detection] 所有 Hook 已就绪\n");
});
```

预期输出：

```
[Port] Hook connect() 完成
[Maps] Hook fgets() 完成
[Strstr] Hook strstr() 完成
[Thread] Hook pthread_setname_np() 完成
[Java] 类枚举完成
[Java] 找到检测函数: com.example.SecurityCheck.isFridaDetected

[Frida Anti-Detection] 所有 Hook 已就绪

[Port] 拦截端口扫描: 27042
[Strstr] 隐藏字符串: frida-agent
[Java] 拦截调用: com.example.SecurityCheck.isFridaDetected
```

第 4 步：针对性绕过

如果你找到了检测函数（从第 1 步）：

```
Java.perform(function () {
    var SecurityCheck = Java.use("com.example.app.SecurityCheck");

    // Hook 检测函数
    SecurityCheck.isFridaDetected.implementation = function () {
        console.log("Bypass isFridaDetected()");
        return false; // 始终返回"未检测到"
    };

    // 拦截其他检测相关函数
    SecurityCheck.checkPort.implementation = function (port) {
        console.log("Bypass checkPort(" + port + ")");
        return false;
    };
});
```

第 5 步：定制 Frida（终极方案）

5.1 修改源码

克隆 Frida：

```
git clone --recurse-submodules https://github.com/frida/frida.git  
cd frida
```

修改特征字符串：

```
# 替换模块名  
find . -type f -exec sed -i 's/frida-agent/system-agent/g' {} +  
find . -type f -exec sed -i 's/frida-server/system-daemon/g' {} +  
  
# 替换线程名  
find . -type f -exec sed -i 's/gum-js-loop/normal-thread/g' {} +  
find . -type f -exec sed -i 's/gmain/worker/g' {} +
```

编译：

```
make
```

原理解析

Frida 检测点分布

检测点分布：

- 1. 网络层
 - 默认端口：27042, 27043
 - D-Bus 协议特征
- 2. 进程层
 - 进程名：frida-server
 - 命令行参数包含 "frida"
- 3. 内存层
 - 模块名：frida-agent.so, frida-gadget.so
 - 线程名：gum-js-loop, gmain, pool-*
 - 函数 Hook：修改系统函数字节码
- 4. 行为层
 - D-Bus 消息、异常系统调用序列

绕过策略对照表

检测方式	绕过策略
进程名检测	重命名 frida-server
/proc/self/maps	Hook <code>fgets()</code> 过滤输出
字符串检测	Hook <code>strstr()</code> 返回 NULL
线程名检测	Hook <code>pthread_setname_np()</code>
Inline Hook 检测	Hook 检测函数本身
多重组合检测	定制 Frida 源码

Hook 时机很重要

App 启动时序:

```
↓
[0.5s] App 静态初始化代码运行
↓
[1s] App onCreate() 开始
↓
[1.5s] ⚠ 反调试检测通常在此运行
↓
[2s] ✗ Attach 模式: Frida 在此时才注入 (太晚)
```

常见问题

问题 1: 绕过脚本不生效

症状: Hook 脚本运行了, 但 App 仍然检测到 Frida

可能原因：

1. Hook 时机太晚

```
# 正确: --no-pause 立即运行  
frida -U -f com.example.app -l bypass.js --no-pause  
  
# 错误: 会暂停等待手动恢复  
frida -U -f com.example.app -l bypass.js
```

1. Native 层检测未覆盖

- 使用 `frida-gadget` 而非 `frida-server` (更早注入)

2. 存在未覆盖的检测点

- 使用 jadx 分析完整的检测逻辑

问题 2: Hook 后 App 崩溃

症状：加载 Hook 脚本后 App 立即崩溃

检查：

1. Hook 的函数签名错误

```
// 检查重载  
Java.use("ClassName").methodName.overloads.forEach(function (o) {  
    console.log(o);  
});
```

1. 返回值类型不匹配

```
// 错误
SomeClass.returnsInt.implementation = function () {
    return "string"; // 类型错误!
};

// 正确
SomeClass.returnsInt.implementation = function () {
    return 0;
};
```

1. Hook 影响了正常功能

- 添加条件判断，只 Hook 特定情况

问题 3: 某些检测绕不过去

症状：尝试了所有方法，仍有检测未绕过

高级对策：

1. 使用 frida-gadget (嵌入式)

```
# 解包 APK
apktool d app.apk

# 将 frida-gadget.so 添加到 lib/
# 修改 AndroidManifest.xml 和 smali 代码加载 gadget
# 参考: https://frida.re/docs/gadget/

# 重新打包
apktool b app -o app_patched.apk
```

1. 使用预编译的定制版

- 社区项目：<https://github.com/hluwa/strongR-frida-android>
- 已重命名所有特征字符串

2. 使用 Docker 编译环境

```
docker run --rm -v $(pwd):/work frida/ci
```

延伸阅读

相关 Recipes

- [Root 检测绕过](#) - 通常与 Frida 检测一起出现
- [SSL Pinning 绕过](#) - 可能也有反 Frida
- [模拟器检测绕过](#) - 多重检测组合

工具深入

- [Frida 完整指南](#)
- [Frida 内部原理](#) - 理解检测原理

案例分析

- [反分析技术案例](#)
- [社交媒体风控](#) - 高级检测对抗

进阶资源

- strongR-frida: <https://github.com/hluwa/strongR-frida-android>
- frida-gadget 文档: <https://frida.re/docs/gadget/>
- 编译 Frida: <https://frida.re/docs/building/>

快速参考

一键绕过脚本

下载通用绕过脚本:

```
# 使用社区维护的绕过脚本
curl -O https://raw.githubusercontent.com/0xdeaf/frida-scripts/master/
raptor_frida_android_bypass.js

# 运行
frida -U -f com.example.app -l raptor_frida_android_bypass.js --no-pause
```

检测点速查表

检测类型	检测位置	绕过方法
进程名	/proc/* cmdline	重命名 + Hook fopen()
模块名	/proc/self/maps	Hook fgets() 过滤
字符串	strstr()	Hook strstr()
线程名	/proc/self/task/* comm	Hook pthread_setname_np()
Java 检测	isFridaDetected()	Hook 检测函数

常用命令

```
# 非标准端口运行 frida-server
adb shell "/data/local/tmp/frida -l 0.0.0.0:8888 &"

# 连接到非标准端口
frida -H 127.0.0.1:8888 -f com.example.app

# Spawn 模式 (重要)
frida -U -f com.example.app -l bypass.js --no-pause

# 列出所有模块 (检查是否有 frida-agent)
frida -U -f com.example.app -e 'Process.enumerateModules()'

# 列出所有线程 (检查线程名)
frida -U -f com.example.app -e 'Process.enumerateThreads()'
```

[R07] Xposed 反调试绕过

R07: 绕过应用的 Xposed 检测

问题场景

你可能遇到以下情况：

1. App 启动即退出：App 启动后弹出“检测到 Xposed 框架，禁止运行”，随即闪退
2. 功能受限：金融/支付类 App 检测到 Xposed 后拒绝提供服务（无法转账、支付）
3. 账号封禁：游戏检测到 Xposed 环境后触发风控，导致封号
4. 分析受阻：需要在 Xposed 环境下分析 App 行为，但被反调试拦截
5. 通用方案失效：已经使用 RootCloak Plus 等通用隐藏模块，但仍被检测到

工具清单

必需工具

- Xposed 框架：EdXposed 或 LSPosed（推荐 LSPosed，更稳定）
- Root 设备：已 Root 的 Android 设备或模拟器（如 Genymotion）
- 目标 APK：需要绕过检测的应用安装包
- Xposed 模块开发环境：Android Studio（用于编写自定义绕过模块）

可选工具

- JEB/Jadx：反编译工具，用于分析 App 的检测代码
- RootCloak Plus：通用 Xposed/Root 隐藏模块（快速测试）
- Hide My Applist：高级应用列表和框架隐藏工具
- MT Manager：Android 文件管理器，查看系统文件

-
- Xposed 源码: EdXposed 源码 (定制化框架需求)

前置条件

开始之前, 请确认:

- 已安装 Xposed 框架: 设备上已刷入 EdXposed 或 LSPosed, 并能正常使用
- 设备已 Root: 拥有 Root 权限, 或使用虚拟化方案 (如 VirtualXposed)
- 能反编译 APK: 会使用 JADX/JEB 查看 Java 代码
- 了解 Xposed Hook 基础: 知道如何编写简单的 Xposed 模块 (可参考 [Xposed Guide](#))
- 了解 Java 反射机制: 理解 `Class.forName()`, `ClassLoader` 等概念

解决方案

第 1 步: 识别检测类型 (15-30 分钟)

首先需要确定 App 使用了哪种检测方法, 这决定了后续的绕过策略。

方法 1: 观察运行行为

运行目标 App, 观察异常行为的时机和特征:

异常时机	可能的检测类型
启动阶段立即崩溃/退出	调用栈检测 (<code>Application.onCreate</code> 中)
特定功能 (登录、支付) 受限	关键方法处的定点检测
延迟几秒后弹出警告	定时器或异步线程中的检测
随机触发	多点分散检测或混淆后的检测

方法 2：静态分析检测代码

使用 JADX 反编译 APK，搜索 Xposed 检测的特征字符串：

```
# 反编译 APK  
jadx -d ./decompiled target.apk  
  
# 搜索 Xposed 相关特征  
cd decompiled  
grep -r "xposed" --include="*.java" .  
grep -r "XposedBridge" --include="*.java" .  
grep -r "de.robv.android" --include="*.java" .  
  
# 搜索检测方法调用  
grep -r "getStackTrace" --include="*.java" .  
grep -r "Class.forName" --include="*.java" .  
grep -r "/proc/self/maps" --include="*.java" .
```

常见检测代码模式：

调用栈检测：

```
// 特征代码  
try {  
    throw new Exception("Xposed Detection");  
} catch (Exception e) {  
    for (StackTraceElement element : e.getStackTrace()) {  
        if (element.getClassName().contains("de.robv.android.xposed")) {  
            // Xposed 检测到!  
            return true;  
        }  
    }  
}
```

类加载检测：

```
try {  
    Class.forName("de.robv.android.xposed.XposedBridge");  
    // 如果没抛异常，说明 Xposed 存在  
    return true;  
} catch (ClassNotFoundException e) {  
    return false;  
}
```

文件系统检测：

```
File xposedJar = new File("/system/framework/XposedBridge.jar");
if (xposedJar.exists()) {
    // Xposed 检测到!
}
```

Native 层检测：

```
// maps 文件检测
FILE* fp = fopen("/proc/self/maps", "r");
// 读取内容并查找 "libxposed_art.so" 或 "XposedBridge"

// 符号地址检测
void* handle = dlopen("libart.so", RTLD_NOW);
void* sym = dlsym(handle, "_ZN3art9ArtMethod6InvokeEPNS_6ThreadEPjjPNS_6JValueEPKc");
// 检查地址是否在非标准内存区域
```

方法 3：选择绕过策略

根据检测类型选择合适的策略：

检测类型	推荐策略	成功率	耗时
Java 层调用栈检测	Hook <code>StackTraceElement.getClassName()</code>	90%	30min
Java 层类加载检测	Hook <code>Class.forName()</code>	85%	20min
文件系统检测	Hook <code>File.exists()</code>	95%	20min
Native 层 maps 检测	Hook <code>fopen()</code> / 定制框架	60%	2-4h
综合检测（多种方法）	定制化 Xposed 框架	80%	4h+

决策树：

检测类型已知？

- └ 是 → 策略 A：编写针对性 Hook 模块
- └ 否 → 快速测试需求或不想写代码
 - └→ 策略 B：使用现成隐藏模块（最快）

第 2 步：策略 A - 编写自定义绕过模块

创建 `AntiXposedDetection.java`：

```
package com.example.antidetect;

import android.os.Bundle;
import de.robv.android.xposed.*;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;
import java.io.File;

public class AntiXposedDetection implements IXposedHookLoadPackage {

    // 修改为你的目标 App 包名
    private static final String TARGET_PACKAGE = "com.target.app";

    @Override
    public void handleLoadPackage(LoadPackageParam lpparam) throws Throwable {
        // 只 Hook 目标 App
        if (!lpparam.packageName.equals(TARGET_PACKAGE)) return;

        XposedBridge.log("[AntiDetect] 开始 Hook " + TARGET_PACKAGE);

        // 绕过调用栈检测
        hookStackTrace();

        // 绕过类加载检测
        hookClassName();

        // 绕过文件系统检测
        hookFileExists();

        // 绕过系统属性检测
        hookSystemProperties();

        XposedBridge.log("[AntiDetect] 所有 Hook 已激活");
    }

    /**
     * 绕过调用栈检测
     * 原理: 修改 StackTraceElement.getClassName() 返回值
     */
    private void hookStackTrace() {
        XposedHelpers.findAndHookMethod(
            StackTraceElement.class,
            "getClassName",
            new XC_MethodHook() {
                @Override
                protected void afterHookedMethod(MethodHookParam param) throws
Throwable {
                    String originalClassName = (String) param.getResult();

                    // 如果类名包含 xposed 特征, 替换为无害系统类
                    if (originalClassName != null &&
                        originalClassName.toLowerCase().contains("xposed")) {
                        param.setResult("com.android.internal.os.ZygoteInit");
                    }
                }
            }
        );
    }
}
```

```
XposedBridge.log("[AntiDetect] 隐藏调用栈: " +
originalClassName);
    }
}
)
);
}

/**
 * 绕过类加载检测
 * 原理: 拦截 Class.forName() 调用, 对 Xposed 类抛出 ClassNotFoundException
 */
private void hookClassForName() {
    XposedHelpers.findAndHookMethod(
        Class.class,
        "forName",
        String.class,
        new XC_MethodHook() {
            @Override
            protected void beforeHookedMethod(MethodHookParam param) throws
Throwable {
                String className = (String) param.args[0];

                // 如果尝试加载 Xposed 相关类, 抛出 ClassNotFoundException
                if (className != null &&
                    (className.contains("xposed") ||
                     className.contains("Xposed") ||
                     className.contains("EdXposed") ||
                     className.contains("LSPosed")))) {
                    param.setThrowable(new ClassNotFoundException(className));
                    XposedBridge.log("[AntiDetect] 阻止加载类: " + className);
                }
            }
        }
    );
}

// 也拦截三参数版本 forName
XposedHelpers.findAndHookMethod(
    Class.class,
    "forName",
    String.class,
    boolean.class,
    ClassLoader.class,
    new XC_MethodHook() {
        @Override
        protected void beforeHookedMethod(MethodHookParam param) throws
Throwable {
            String className = (String) param.args[0];
            if (className != null &&
                className.toLowerCase().contains("xposed")) {
                param.setThrowable(new ClassNotFoundException(className));
            }
        }
    }
);
```

```
        }
    );
}

/***
 * 绕过文件系统检测
 * 原理: 修改 File.exists() 返回值, 隐藏 Xposed 特征文件
 */
private void hookFileExists() {
    XposedHelpers.findAndHookMethod(
        File.class,
        "exists",
        new XC_MethodHook() {
            @Override
            protected void afterHookedMethod(MethodHookParam param) throws
Throwable {
                File file = (File) param.thisObject;
                String path = file.getAbsolutePath();

                // Xposed 特征文件列表
                String[] xposedPaths = {
                    "XposedBridge",
                    "xposed",
                    "de.robv.android.xposed",
                    "EdXposed",
                    "LSPosed",
                    "libxposed",
                    "libedxposed",
                    "liblspd"
                };
                // 检查路径是否包含特征
                for (String keyword : xposedPaths) {
                    if (path.contains(keyword)) {
                        param.setResult(false); // 伪装文件不存在
                        XposedBridge.log("[AntiDetect] 隐藏文件: " + path);
                        return;
                    }
                }
            }
        );
}

/***
 * 绕过系统属性检测
 * 原理: 拦截 System.getProperty() 调用
 */
private void hookSystemProperties() {
    XposedHelpers.findAndHookMethod(
        System.class,
        "getProperty",
        String.class,
```

```
new XC_MethodHook() {
    @Override
    protected void afterHookedMethod(MethodHookParam param) throws
Throwable {
        String key = (String) param.args[0];

        // VirtualXposed 等会设置特殊属性
        if (key != null &&
            (key.contains("xposed") ||
             key.contains("vxp") ||
             key.equals("ro.build.version.xposed")))) {
            param.setResult(null); // 返回null
            XposedBridge.log("[AntiDetect] 隐藏系统属性: " + key);
        }
    }
};

}
}
```

项目结构:

```
AntiXposedDetection/
├── app/
│   └── src/main/
│       ├── AndroidManifest.xml
│       ├── assets/
│       │   └── xposed_init # 入口类声明
│       │       └── java/com/example/antidetect/
│       │           └── AntiXposedDetection.java
└── build.gradle
```

AndroidManifest.xml:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.antidetect">  
  
    <application  
        android:allowBackup="true"  
        android:label="Anti Xposed Detection"  
        android:icon="@mipmap/ic_launcher">  
  
        <!-- Xposed 模块声明 -->  
        <meta-data  
            android:name="xposedmodule"  
            android:value="true" />  
        <meta-data  
            android:name="xposeddescription"  
            android:value="Hide Xposed from target app" />  
        <meta-data  
            android:name="xposedminversion"  
            android:value="54" />  
    </application>  
</manifest>
```

编译和安装：

```
# 1. 编译模块  
./gradlew assembleRelease  
  
# 2. 安装到设备  
adb install app/build/outputs/apk/release/app-release.apk  
  
# 3. 在 LSPosed/EdXposed 中激活模块  
# - 打开 LSPosed Manager  
# - 模块 → 找到 "Anti Xposed 检测"  
# - 勾选启用，并在作用域中添加目标 App  
# - 重启目标 App  
  
# 4. 查看日志验证  
adb logcat -s Xposed:V | grep AntiDetect
```

第 3 步：策略 B - 使用现成隐藏模块

1. Hide My Applist（最强大，推荐）

- 下载：[GitHub](#)
- 功能：隐藏应用列表、Xposed 框架、Root

- 支持黑白名单、模板系统

2. XposedChecker Bypass

- 专门针对 XposedChecker 这类检测工具
 - 覆盖常见检测点

3. RootCloak Plus (老牌模块)

- 同时隐藏 Root 和 Xposed
 - 配置简单，但对新型检测效果较差

使用步骤（以 Hide My Applist 为例）：

```
# 1. 下载并安装
# 从 GitHub 发布页下载最新 APK
adb install HideMyApplist.apk

# 2. 在 LSPosed 中激活
# LSPosed 管理器 → 模块 → Hide My Applist → 勾选启用

# 3. 配置隐藏规则
# 打开 Hide My Applist App
# → 模板管理 → 新建模板
# → 选择隐藏内容：
#   黑名单模式（隐藏 Xposed 相关）
#   隐藏 Xposed 模块
#   隐藏系统框架
# → 应用管理 → 选择目标 App → 应用模板

# 4. 重启目标 App
adb shell am force-stop com.target.app
adb shell am start -n com.target.app/.MainActivity
```

第 4 步：策略 C - 定制 Xposed 框架

当 Hook 绕过无效时，可能需要从源码层面修改 Xposed 特征。

修改目标：

- 包名：`de.robv.android.xposed` → `com.myfw.custom`
- 类名：`XposedBridge` → `CustomBridge`

- 文件名: `libxposed_art.so` → `libcustom_art.so`
- 系统属性: `persist.xposed.*` → `persist.myfw.*`

详细步骤:

1. 获取 Xposed 源码:

```
# 克隆 EdXposed 源码 (推荐, 比原版 Xposed 更活跃)
git clone --recursive https://github.com/ElderDrivers/EdXposed
cd EdXposed

# 也可以克隆 LSPosed (更现代实现)
git clone --recursive https://github.com/LSPosed/LSPosed
```

2. 修改特征字符串 (创建脚本 `rename_xposed.sh`) :

```
#!/bin/bash

# 全局替换 Xposed 特征为自定义名称
OLD_PACKAGE="de.robv.android.xposed"
NEW_PACKAGE="com.myframework.custom"

OLD_CLASS="Xposed"
NEW_CLASS="Custom"

OLD_LIB="xposed"
NEW_LIB="myfw"

echo "开始替换 Xposed 特征..."

# 1. 替换包名
echo "替换包名..."
find . -type f \(-name "*.java" -o -name "*.cpp" -o -name "*.h"\) \
-exec sed -i "s/$OLD_PACKAGE/$NEW_PACKAGE/g" {} +

# 2. 替换类名
echo "替换类名..."
find . -type f -name "*.java" \
-exec sed -i "s/${OLD_CLASS}Bridge/${NEW_CLASS}Bridge/g" {} +
find . -type f -name "*.java" \
-exec sed -i "s/${OLD_CLASS}Helpers/${NEW_CLASS}Helpers/g" {} +

# 3. 重命名文件
echo "重命名文件..."
find . -name "*Xposed*" | while read file; do
    newfile=$(echo "$file" | sed "s/Xposed/Custom/g")
    mv "$file" "$newfile" 2>/dev/null
done

# 4. 替换库文件名
echo "替换Native 库名..."
find . -type f \(-name "*.cpp" -o -name "*.mk" -o -name "CMakeLists.txt"\) \
-exec sed -i "s/lib${OLD_LIB}/lib${NEW_LIB}/g" {} +

# 5. 替换系统属性名
echo "替换系统属性..."
find . -type f \(-name "*.cpp" -o -name "*.java"\) \
-exec sed -i "s/persist.xposed/persist.myfw/g" {} + \
-exec sed -i "s/ro.xposed/ro.myfw/g" {} +

echo "特征替换完成!"
echo "请手动检查以下文件是否正确:"
echo "  - AndroidManifest.xml"
echo "  - module.prop (Magisk 模块配置)"
echo "  - build.gradle"
```

3. 修改 module.prop (Magisk 模块配置) :

```
id=custom_xposed  
name=Custom Framework (Xposed)  
version=v1.0.0  
versionCode=1  
author=YourName  
description=Customized Xposed Framework with renamed signatures  
minMagisk=21000
```

4. 编译定制版框架：

```
# 执行替换  
./rename_xposed.sh  
  
# 编译 (以 EdXposed 为例)  
cd EdXposed  
./gradlew :edxp-core:buildAll  
  
# 输出位于 out/edxp-core/release/  
# 获取一个 .zip 文件, 可在 Magisk 中刷入
```

5. 安装定制版框架：

```
# 方法 1: 通过 Magisk 管理器安装  
# 打开 Magisk 管理器 → 模块 → 从本地安装 → 选择 ZIP  
  
# 方法 2: 通过 TWRP Recovery 刷入 (如有)  
# adb reboot recovery  
# 在 TWRP 中选择 Install → 选择 ZIP → 滑动确认  
  
# 重启设备  
adb reboot
```

第 5 步：验证绕过效果

验证清单：

- 目标 App 正常运行：不再弹出检测警告，不闪退
- Xposed Hook 依然生效：你的 Hook 模块能正常 Hook 目标方法
- 检测工具显示干净：使用 XposedChecker 等工具测试，显示未检测到

验证方法：

1. 编写测试 Xposed 模块:

```
// 验证拦截是否生效
XposedHelpers.findAndHookMethod(
    "com.target.app.MainActivity",
    lpparam.classLoader,
    "onCreate",
    Bundle.class,
    new XC_MethodHook() {
        @Override
        protected void afterHookedMethod(MethodHookParam param) throws Throwable {
            XposedBridge.log("Hook 成功! App 未检测到 Xposed");
        }
    }
);
```

2. 查看日志:

```
# 查看 App 是否有检测相关日志
adb logcat | grep -i "xposed\|detect\|security\|check"

# 应该看到:
# - Hook 成功日志
# - 没有检测相关错误日志
```

3. 使用 XposedChecker:

```
# 安装 XposedChecker
adb install XposedChecker.apk

# 运行并查看结果
# 如果绕过成功, 应该显示 "Xposed: Not Detected"
```

4. 测试目标 App 功能:

完整功能测试流程:

- 启动 App (观察是否崩溃)
- 登录账号 (观察是否被拒绝)
- 测试核心业务 (支付/游戏/查看敏感信息)
 - 观察是否触发风控或异常

原理解析

Xposed 检测原理

1. 调用栈特征：

当 Xposed Hook 一个方法时，实际的调用链是：

```
App.targetMethod()  
↓  
XposedBridge.handleHookedMethod()  
↓  
原始方法/替换方法
```

App 可以通过 `Thread.currentThread().getStackTrace()` 获取调用栈，如果发现 `de.robv.android.xposed` 相关类，就说明 Xposed 存在。

2. 类加载特征：

Xposed 在 Zygote 进程中注入，会加载 `XposedBridge` 等类到每个 App 进程。App 可以尝试 `Class.forName("de.robv.android.xposed.XposedBridge")`，如果成功加载，就说明 Xposed 存在。

3. 文件系统特征：

Xposed 需要在系统中安装文件：

- `/system/framework/XposedBridge.jar`
- `/system/lib/libxposed_art.so` 或 `/system/lib64/libxposed_art.so`

4. 内存映射特征：

进程的内存映射 (`/proc/self/maps`) 中会出现 Xposed 相关的库。

绕过原理

关键技术点：

1. 拦截检测方法的执行：在检测代码执行前 Hook，修改其行为
2. 修改返回值：让检测方法总是返回"未检测到"的结果
3. 过滤特征字符串：将包含 "xposed" 的字符串替换为无害字符串
4. 阻止异常抛出：对于 `Class.forName()` 这类检测，主动抛出 `ClassNotFoundException`

示例：调用栈检测的绕过原理

```
Hook 前 (检测成功)：  
App 调用 getStackTrace()  
→ 返回 [MainActivity, XposedBridge, ZygoteInit, ...]  
→ App 发现 "XposedBridge"  
→ 检测成功 (Xposed 存在)  
  
Hook 后 (绕过检测)：  
App 调用 getStackTrace()  
→ 返回 [MainActivity, XposedBridge, ZygoteInit, ...]  
→ 我们 Hook 拦截 getClassNames()  
→ 将 "XposedBridge" 替换为 "ZygoteInit"  
→ App 只看到 [MainActivity, ZygoteInit, ZygoteInit, ...]  
→ 检测失败 (未发现 Xposed)
```

定制框架效果

检测方式	检测代码	定制后	结果
类加载检测	<code>Class.forName("de.robv...XposedBridge")</code>	包名改为 <code>com.myfw...CustomBridge</code>	检测失败
文件检测	<code>/system/framework/XposedBridge.jar</code>	文件名改为 <code>CustomBridge.jar</code>	检测失败
maps检测	搜索 <code>libxposed_art.so</code>	库名改为 <code>libcusom_art.so</code>	检测失败

缺点：

- 维护成本高，需要跟随官方 Xposed 更新
- 编译过程复杂，需要配置 Android NDK
- 部分依赖原版 Xposed API 的模块可能不兼容

常见问题

问题 1：Hook 模块激活后，App 仍然检测到 Xposed

可能原因：

1. Hook 时机太晚：App 在 `Application.onCreate()` 之前就检测了

2. Native 层检测: App 使用 JNI 检测, Java Hook 无法拦截
3. 遗漏的检测点: App 使用了你没有覆盖的检测方法
4. Hook 作用域未配置: LSPosed 中未将目标 App 加入作用域

解决方案:

方案 1: 提前 Hook 时机

```
// 在 Application.attachBaseContext() 中提前拦截
XposedHelpers.findAndHookMethod(
    "com.target.app.MyApplication",
    lpparam.classLoader,
    "attachBaseContext",
    Context.class,
    new XC_MethodHook() {
        @Override
        protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
            // 在 Application 初始化前就拦截所有检测方法
            hookAllDetectionMethods(lpparam);
            XposedBridge.log("[AntiDetect] Early hooks installed");
        }
    });
);
```

方案 2: 结合 Frida 处理 Native 层

```
// Frida Script: Hook fopen() 拦截 maps 文件读取
Interceptor.attach(Module.findExportByName("libc.so", "fopen"), {
    onEnter: function(args) {
        var path = Memory.readUtf8String(args[0]);
        if (path === "/proc/self/maps") {
            console.log("[*] fopen() called for /proc/self/maps");
            // 重定向到一个干净的 maps 文件
            args[0] = Memory.allocUtf8String("/data/local/tmp/fake_maps");
        }
    }
});
```

方案 3: 使用 Xposed 调用 Native Hook

```
// 调用 native 方法做 inline hook
nativeHookFopen();
```

方案 4：全面分析检测代码

```
cd decompiled  
  
# 搜索所有可能检测模式  
grep -rn "getStackTrace\|forName\|/proc/self/maps\|XposedBridge\|exists()" . \  
--include="*.java" > detection_points.txt
```

方案 5：检查 LSPosed 作用域

```
adb shell "su -c 'ls /data/adb/lspd/config/modules/'"  
# 应该看到你的模块 ID  
  
# 检查作用域配置  
# LSPosed 管理器 → 模块 → 你的模块 → 应用作用域  
# 确保目标 App 已勾选
```

问题 2：定制框架编译失败

可能原因：

- META-INF 目录下的脚本缺失
- 编译过程中出错，生成的 ZIP 损坏

解决方案：

方案 1：检查 ZIP 结构

```
# 解压查看结构  
unzip -l edxp-custom.zip  
  
# 标准 Magisk 模块结构:  
# META-INF/  
#   com/google/android/  
#     update-binary      # 安装脚本  
#     updater-script    # 空文件即可  
#   module.prop        # 模块配置  
#   system/            # 系统文件  
#   framework/  
#     CustomBridge.jar  
#   lib64/  
#     libcustom_art.so  
#   riru/              # Riru 相关 (如使用 Riru)
```

方案 2：手动创建 module.prop

```
id=custom_xposed  
name=Custom Xposed Framework  
version=v1.0.0  
versionCode=100  
author=YourName  
description=Customized Xposed with renamed signatures  
  
# 可选字段  
minMagisk=21000  
maxMagisk=99999
```

方案 3：手动打包

```
mkdir -p magisk_module/system/framework
mkdir -p magisk_module/system/lib64

# 复制文件
cp update-binary magisk_module/META-INF/com/google/android/
touch magisk_module/META-INF/com/google/android/updater-script
cp module.prop magisk_module/
cp CustomBridge.jar magisk_module/system/framework/
cp libcustom_art.so magisk_module/system/lib64/

# 打包 (注意: 必须在模块目录内打包)
cd magisk_module
zip -r ../custom-xposed-magisk.zip .
cd ..

# 推送并安装
adb push custom-xposed-magisk.zip /sdcard/
# 在 Magisk Manager 中安装
```

方案 4：使用 TWRP 刷入

```
adb reboot recovery
# 在 TWRP 中: Install → 选择 ZIP → 滑动确认
```

问题 3：Hook 后 App 功能异常或崩溃

可能原因：

- Hook 范围太广，影响了正常功能
- 返回值类型不匹配
- Hook 的方法签名不正确

解决方案：

方案 1：精准 Hook，缩小作用范围

错误示范：全局 Hook

```
// 这样会影响所有类加载，包括正常业务
XposedHelpers.findAndHookMethod(Class.class, "forName", String.class, ...);
```

正确示范：针对性 Hook

```
// 只 Hook 已知的检测类
XposedHelpers.findAndHookMethod(
    "com.target.app.security.SecurityChecker",
    lpparam.classLoader,
    "checkXposed",
    new XC_MethodReplacement() {
        @Override
        protected Object replaceHookedMethod(MethodHookParam param) {
            XposedBridge.log("[AntiDetect] checkXposed() blocked");
            return false; // 返回"未检测到"
        }
    }
);
```

方案 2：添加条件判断

```
@Override
protected void afterHookedMethod(MethodHookParam param) throws Throwable {
    String className = (String) param.getResult();

    // 只过滤 Xposed 相关，不影响其他类
    if (className != null && className.toLowerCase().contains("xposed")) {
        param.setResult("android.app.Activity");
    }
    // 其他情况保持原样，不做修改
}
```

方案 3：逐个启用 Hook 测试

```
@Override
public void handleLoadPackage(LoadPackageParam lpparam) throws Throwable {
    // 一次只启用一个拦截，测试是否导致崩溃
    hookStackTrace(); // 测试: OK
    // hookClassForName(); // 暂时注释掉
    // hookFileExists(); // 暂时注释掉

    // 逐个启用并测试，找出导致崩溃的拦截
}
```

方案 4：确保返回值类型正确

```
// 确保返回值类型匹配
@Override
protected void afterHookedMethod(MethodHookParam param) {
    // 如果原方法返回 boolean, 你也必须返回 boolean
    param.setResult(false); // 正确
    // param.setResult("false"); // 错误! 类型不匹配
}
```

问题 4：模块在 LSPosed 中不显示

可能原因：

- `assets/xposed_init` 文件缺失或路径错误
- `AndroidManifest.xml` 中缺少 Xposed 元数据声明
- 模块入口类的包名/类名与 `xposed_init` 中不一致
- LSPosed 缓存未刷新

解决方案：

方案 1：检查 `assets/xposed_init` 文件

```
# 确认文件存在
unzip -l app-release.apk | grep xposed_init
# 应该看到: assets/xposed_init

# 检查内容 (必须是完整类名, 无文件扩展名)
unzip -p app-release.apk assets/xposed_init
# 输出应该是: com.example.antidetect.AntiXposedDetection
```

方案 2：确认文件路径正确

正确路径：
`app/src/main/assets/xposed_init # 正确`

错误路径：
`app/assets/xposed_init # 错误`

方案 3：检查 `AndroidManifest.xml`

```
<!-- 必须有这三个 meta-data -->
<meta-data
    android:name="xposedmodule"
    android:value="true" />
<meta-data
    android:name="xposeddescription"
    android:value="Hide Xposed from detection" />
<meta-data
    android:name="xposedminversion"
    android:value="54" />
</application>
```

方案 4：确认入口类存在

```
# 检查入口类是否存在
ls -l ./decompiled/com/example/antidetect/AntiXposedDetection.java

# 确认类实现了 IXposedHookLoadPackage 接口
grep "implements IXposedHookLoadPackage" \
./decompiled/com/example/antidetect/AntiXposedDetection.java
```

方案 5：刷新 LSPosed 缓存

```
# 清除 LSPosed 缓存
adb shell "su -c 'rm -rf /data/adb/lspd/cache/*'"

# 重新安装
adb install app-release.apk

# 重启 LSPosed (或重启设备)
adb shell "su -c 'killall -9 com.android.systemui'"
# 或
adb reboot

# 打开 LSPosed 管理器，应该能看到模块
```

延伸阅读

相关 Recipes

- [逆向工程工作流](#) - 完整的逆向分析流程

工具深入

- [Xposed 使用指南](#) - Xposed 框架基础使用
- [Xposed 内部原理](#) - Xposed 工作机制详解
- [Frida 使用指南](#) - Frida 与 Xposed 协同使用

参考资料

- [Android 沙箱实现](#) - 虚拟化环境中使用 Xposed
- [ART 运行时](#) - 理解 Xposed 如何修改 ART

案例分析

- [反分析技术案例](#) - 综合反分析技术案例
- [社交媒体风控](#) - 社交应用的 Xposed 检测

快速参考

Xposed 检测方法速查表

检测类型	检测层级	特征代码	绕过方法	Hook 目标
调用栈检测	Java	<code>getStackTrace() + contains("xposed")</code>	Hook 返回值过滤	<code>StackTraceElement</code>
类加载检测	Java	<code>Class.forName("XposedBridge")</code>	抛出 ClassNotFoundException	<code>Class.forName</code>
已加载类检测	Java	<code>ClassLoader.loadClass(...)</code>	同上	<code>ClassLoader</code>
文件检测	Java	<code>new File("/system/.../XposedBridge.jar").exists()</code>	返回 false	<code>File.exists()</code>
maps 检测	Native	<code>fopen("/proc/self/maps") + strstr("libxposed")</code>	Hook fopen 或定制框架	<code>libc.fopen()</code>
系统属性检测	Java/ Native	<code>System.getProperty("vxp_...")</code>	返回 null	<code>System.getProperty</code>
符号地址检测	Native	<code>dlsym(...)</code> 检查地址异常	定制框架	N/A (需源码修改)

完整绕过模块模板（一键使用）

保存为 `AntiXposedBypass.java`，修改包名和目标 App 即可使用：

```
package com.example.antidetect;

import de.robv.android.xposed.*;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;
import java.io.File;

public class AntiXposedBypass implements IXposedHookLoadPackage {

    private static final String TARGET = "com.target.app"; // 改为你的目标App

    @Override
    public void handleLoadPackage(LoadPackageParam lpparam) throws Throwable {
        if (!lpparam.packageName.equals(TARGET)) return;

        XposedBridge.log("[Bypass] Hooking " + TARGET);

        // 1. 绕过调用栈检测
        XposedHelpers.findAndHookMethod(StackTraceElement.class, "getClassName",
            new XC_MethodHook() {
                @Override
                protected void afterHookedMethod(MethodHookParam param) throws
Throwable {
                    String name = (String) param.getResult();
                    if (name != null && name.toLowerCase().contains("xposed")) {
                        param.setResult("com.android.internal.os.ZygoteInit");
                    }
                }
            });
    }

    // 2. 绕过类加载检测
    XposedHelpers.findAndHookMethod(Class.class, "forName", String.class,
        new XC_MethodHook() {
            @Override
            protected void beforeHookedMethod(MethodHookParam param) throws
Throwable {
                String cls = (String) param.args[0];
                if (cls != null && cls.toLowerCase().contains("xposed")) {
                    param.setThrowable(new ClassNotFoundException(cls));
                }
            }
        });
}

// 3. 绕过文件检测
XposedHelpers.findAndHookMethod(File.class, "exists",
    new XC_MethodHook() {
        @Override
        protected void afterHookedMethod(MethodHookParam param) throws
Throwable {
            String path = ((File) param.thisObject).getAbsolutePath();
            if (path.toLowerCase().contains("xposed") ||
                path.toLowerCase().contains("edxposed") ||
                path.toLowerCase().contains("lspd")) {
```

```
        param.setResult(false);
    }
}
});

XposedBridge.log("[Bypass] All hooks activated");
}

}
```

常用命令速查

```
# ===== 环境检查 =====

# 检查 Xposed 框架状态
adb shell "su -c 'ls -l /system/framework/Xposed'"
adb shell "su -c 'ps -A | grep xposed'"

# 查看已安装的 Xposed 模块
adb shell "su -c 'ls /data/app/ | grep -i xposed'"

# 检查 LSPosed 状态
adb shell "su -c 'ls -l /data/adb/lspd/'"

# ===== 日志调试 =====

# 查看 Xposed 框架日志
adb logcat -s Xposed:V

# 查看模块日志 (假设模块标签为 AntiDetect)
adb logcat | grep AntiDetect

# 查看 App 检测相关日志
adb logcat | grep -iE "detect|xposed|security|check"

# 清空日志并实时查看
adb logcat -c && adb logcat -v time

# ===== 模块管理 =====

# 编译 Xposed 模块
./gradlew assembleDebug    # Debug 版
./gradlew assembleRelease # Release 版

# 安装模块
adb install app/build/outputs/apk/debug/app-debug.apk

# 卸载模块
adb uninstall com.example.antidetect

# 重启目标 App (应用更改)
adb shell am force-stop com.target.app
adb shell am start -n com.target.app/.MainActivity

# ===== 定制框架 =====

# 推送自定义框架到设备
adb push EdXposed-custom.zip /sdcard/

# 在 Magisk 中安装 (命令行方式)
adb shell "su -c 'magisk --install-module /sdcard/EdXposed-custom.zip'"

# 重启设备
adb reboot
```

```
# ===== 测试验证 =====

# 安装 XposedChecker 测试工具
adb install XposedChecker.apk

# 运行目标 App 并观察行为
adb shell am start -n com.target.app/.MainActivity

# 抓取崩溃日志
adb logcat -b crash

# 检查进程内存映射 (查找 Xposed 特征)
adb shell "su -c 'cat /proc/$(pidof com.target.app)/maps | grep -i xposed'"
```

常见检测代码模式

```
// ===== Java 层检测模式 =====

// 模式 1: 调用栈检测
if (element.getClassName().contains("xposed")) { /* 检测到 */ }

// 模式 2: 异常调用栈检测
try { throw new Exception(); } catch (Exception e) {
    for (StackTraceElement elem : e.getStackTrace()) { /* 检查 */ }
}

// 模式 3: 类加载检测
Class.forName("de.robv.android.xposed.XposedBridge");

// 模式 4: ClassLoader 检测
ClassLoader.getSystemClassLoader().loadClass("de.robv.android.xposed.XposedHelpers");

// 模式 5: 文件检测
new File("/system/framework/XposedBridge.jar").exists()

// 模式 6: 系统属性检测
System.getProperty("vxp_forbid_status")
System.getProperty("ro.xposed.version")

// ===== Native 层检测模式 =====

// 模式 7: maps 文件检测 (C/C++)
FILE* fp = fopen("/proc/self/maps", "r");
// 然后搜索 "xposed" 或 "libxposed"

// 模式 8: dlopen 检测
void* handle = dlopen("libxposed_art.so", RTLD_NOW);
if (handle != NULL) { /* 检测到 */ }
```

搜索检测代码的关键词

```
xposed
XposedBridge
de.robv.android.xposed
getStackTrace
Class.forName
/proc/self/maps
libxposed
vxp_forbid
ro.xposed
EdXposed
LSPosed
```

App 类型与推荐策略

App 类型	常见检测方式	推荐策略	成功率
普通应用（社交、工具）	Java 层调用栈检测	策略 A：通用 Hook 模块	95%
金融 App（银行、支付）	Java + Native 综合检测	策略 C：定制框架 + Hook	70%
大型游戏	Native 层 + 定时检测	策略 C：定制框架	60%
安全类 App（VPN、杀毒）	深度检测 + 完整性校验	策略 C + 虚拟化	50%
小众 App	简单检测或无检测	策略 B：现成模块	99%

成功率说明：

- 95%+：通用方法即可绕过
- 70-90%：需要针对性编写 Hook
- 50-70%：需要定制框架或多种技术组合
- <50%：可能需要虚拟化、系统级修改等高级技术

[R08] 验证码绕过技术

R08: 验证码绕过技术：滑块与点选篇

滑块和点选（或称图标）验证码是现代 Web 应用中用于区分人类用户和自动化程序（机器人）的常见手段。与传统的字符输入验证码相比，它们更注重于分析用户的“行为特征”。本文旨在详细介绍绕过这两类验证码的主流技术和核心思想。

目录

- 验证码绕过技术：滑块与点选篇
 - 目录
 - 验证码核心机制
 - 滑块验证码
 - 点选验证码
 - 绕过策略一：模拟人类行为
 - 步骤 1: 目标识别 (计算机视觉)
 - 步骤 2: 轨迹模拟 (核心关键)
 - 第三方打码平台
 - 绕过策略三：寻找逻辑漏洞
 - 防御与对抗的演进

验证码核心机制

滑块验证码

- 目标: 用户需要将滑块拖动到背景图的缺口位置。

- 验证重点:

1. 结果准确性: 滑块最终停留的位置是否在缺口的目标容差范围内。
2. 行为可信度: (更重要) 用户的鼠标轨迹是否像人。一个由程序生成的、匀速的、完美的直线轨迹几乎肯定会被判定为机器人。

点选验证码

- 目标: 根据提示, 按顺序点击图片中的一个或多个汉字、图标或物体。

- 验证重点:

1. 识别准确性: 是否能正确识别并点击目标。
 2. 行为可信度: 点击的坐标、间隔时间、鼠标移动轨迹是否自然。
-

绕过策略一：模拟人类行为

这是最主流、最根本的绕过方法, 其核心是尽可能地模仿人类操作的不完美性。

步骤 1: 目标识别 (计算机视觉)

在模拟操作之前, 程序需要先像人一样"看懂"验证码。

- 对于滑块验证码 (缺口识别):

- 常用库: OpenCV (Python)。

- 方法一: 边缘检测:

1. 获取带缺口的背景图和不带缺口的完整背景图 (通常可以从网络请求中找到)。
2. 使用 Canny 等边缘检测算法分别处理两张图片。
3. 对比两张图的边缘差异, 差异最显著的区域就是缺口的位置。

- 方法二: 模板匹配:

1. 从网络请求或页面元素中获取到独立的"滑块"图片。
-

2. 将滑块图片作为"模板", 在带缺口的背景图上进行模板匹配 (`cv2.matchTemplate`)。匹配度最高的地方就是缺口的起始 X 坐标。

- 对于点选验证码 (目标识别):
- 如果目标是固定的文字或图标, 可以采用与滑块类似的模板匹配方法。
- 如果目标是变化的、复杂的, 例如"请点击图中所有的公交车", 则需要依赖更高级的机器学习模型 (见策略二) 。

步骤 2: 轨迹模拟 (核心关键)

这是整个绕过过程的灵魂。一个好的轨迹模拟算法需要考虑以下几点, 以欺骗服务器端的行为分析模型:

- 非线性移动: 绝对不能是 (x_1, y_1) 到 (x_2, y_2) 的直线。路径需要是带有弧度的曲线。
- 变速移动: 模拟人类操作的肌肉控制, 轨迹应该是"慢-快-慢"的模式。
- 初段加速: 初始移动速度较慢。
- 中段匀速/加速: 中间过程速度加快。
- 末段减速: 接近目标时, 速度会显著减慢, 进行微调。
- 随机抖动: 在主轨迹的基础上, 叠加微小的、随机的 Y 轴 (有时也包括 X 轴) 偏移, 模拟手部自然的抖动。
- 超越与回退: 有时可以模拟"拖过头了一点点, 再往回拉"的行为, 这会极大地增加轨迹的可信度。
- 停顿: 在拖动过程中可以加入短暂的、随机时长的停顿。
- 实现示例 (伪代码):

```
def generate_human_like_track(target_distance):
    track = []
    current_pos = 0
    # Movement pattern: accelerate first, then decelerate
    while current_pos < target_distance:
        # 1. Calculate movement step size for current phase (non-uniform speed)
        if current_pos < target_distance * 0.7:
            step = random.uniform(2, 4) # Acceleration phase
        else:
            step = random.uniform(0.5, 2) # Deceleration and fine-tuning phase

        # 2. Add random jitter
        y_offset = random.uniform(-1, 1)

        # 3. Record trajectory point
        track.append((step, y_offset, random.uniform(10, 50))) # (x step, y offset, time
        interval ms)
        current_pos += step

    # (Optional) Add "overshoot and pull back" trajectory points
    # ...
    return track
```

- 适用场景：需要从一张大图中识别并定位多个不规则物体的点选验证码（例如，“选出所有的红绿灯”）。
- 方法：

1. 数据标注：收集大量的验证码图片，并手动标注出需要识别的物体（如“红绿灯”、“公交车”）。
2. 模型训练：使用标注好的数据集训练一个目标检测模型，如 YOLOv5 或 SSD。
3. 推理：在实际绕过时，将验证码图片输入到训练好的模型中，模型会返回所有识别到的目标的位置坐标。
4. 后续操作：拿到坐标后，再结合上一节的“轨迹模拟”方法去点击。

第三方打码平台

- 概念：将识别验证码这一专业任务外包给第三方服务。这些平台背后通常是大量的人工或者更强大的 AI 模型。
- 代表服务：2Captcha, Anti-Captcha 等。

- 工作流程:

1. 注册并充值。
 2. 通过 API 将验证码图片（或任务描述）发送给平台。
 3. 平台返回结果（如滑块的 X 坐标，或点选目标的坐标序列）。
 4. 你的程序拿到结果后，再执行后续的模拟操作。
- 优点: 成功率极高，能解决几乎所有类型的验证码，无需自己维护复杂的识别模型。
 - 缺点: 需要付费，存在隐私和安全风险（将数据发给第三方），有网络延迟。
-

绕过策略三：寻找逻辑漏洞

在投入大量精力编写复杂的模拟和识别代码前，先尝试寻找“捷径”是一种高性价比的策略。

- 分析前端 JS: 仔细审查页面的 JavaScript 文件，特别是与验证码相关的逻辑。有时可能会发现：
 - 答案硬编码或弱加密: 缺口位置、目标坐标等信息以明文或简单加密的方式存在于前端代码中。
 - 可预测的随机数: 用于生成验证码的随机种子或算法过于简单，可以被预测。
 - API 漏洞:
 - 验证绕过: 尝试直接调用提交表单的 API，但不带验证码相关的参数，看后端是否强制校验。
 - Token 重放: 成功通过一次验证后，获取到的 `session_token` 或 `captcha_id` 是否可以被多次重用。
-

防御与对抗的演进

验证码提供商也在不断进化，以对抗上述绕过技术：

- 环境检测：检测 WebDriver 特征（如 `navigator.webdriver` 标志）、浏览器指纹、字体、分辨率等。
- 更复杂的行为分析：不仅仅是轨迹，还会分析点击压力、滚轮行为、鼠标加速度等更深层次的生物特征。
- 图像干扰：在验证码图片上增加干扰线、噪点、形变、颜色抖动，增加 CV 识别难度。
- 无感验证 (reCAPTCHA v3)：完全在后台根据用户的综合行为评分，分数过低时才弹出挑战。

[R09] 应用加固识别

R09: 主流应用加固厂商及其特征识别

Android 应用加固是一种保护 App 不被轻易逆向、篡改或攻击的技术手段。对于逆向工程师而言，在开始分析一个 App 之前，首要任务就是识别出它所使用的加固厂商，因为不同的加固方案需要不同的脱壳和分析策略。

本指南旨在系统性地总结中国市场主流加固厂商的静态特征“指纹”，帮助分析人员快速识别目标。

目录

- 1. 通用识别思路
- 2. 主流厂商特征详解
 - 主流应用加固厂商及其特征识别
 - 目录
 - 通用识别思路
 - 主流厂商特征详解
 - 榜榜安全 (Bangcle)
 - 360 加固 (Qihoo 360)
 - 腾讯乐固 (Tencent Legu)
 - 网易易盾 (Netease Yidun)
 - 爱加密 (Ijiami)
 - 快速识别摘要表
 - 1. 快速识别摘要表

通用识别思路

识别加固厂商通常遵循以下静态分析路径：

1. 检查 DEX 文件：解压 APK，查看主 `classes.dex` 文件的大小。如果它非常小（通常小于 1MB），而 APK 本身体积很大，那么它很可能是一个“壳”，负责加载真正的、被加密隐藏起来的 DEX。
2. 检查 SO 库：查看 `lib/[arch]/` 目录下的 `.so` 文件列表。加固厂商通常会放入带有自身品牌标识的 SO 库，这是最明显的特征。
3. 检查 `assets` 目录：很多加固方案会将加密后的 DEX 文件、配置文件或其他组件放入 `assets` 目录。
4. 检查 `AndroidManifest.xml`：加固方案通常会用自己的代理 `Application` 类替换掉原始的 `Application` 类。检查 `application` 标签下的 `android:name` 属性，可以找到代理类的名字，其包名往往暴露厂商信息。

主流厂商特征详解

梆梆安全 (Bangle)

梆梆是最早期的加固厂商之一，特征相对明显。

- SO 库特征：
 - `libSecShell.so`
 - `libsecexe.so`
 - `libsecmain.so`
- Java 层特征：
 - 代理 Application 包名：`com.bangcle.protect` 或 `com.secshell.shell`。
- `assets` 目录特征：
 - 可能会有 `bangcle_classes.jar` 或类似命名的加密 DEX 文件。

-
- 其他:
 - `AndroidManifest.xml` 的 `meta-data` 中可能会包含原始 Application 的信息。

360 加固 (Qihoo 360)

360 加固非常普遍，其特征也广为人知。

- SO 库特征:
 - `libjiagu.so`
 - `libprotectClass.so`
 - `libjiagu_x86.so` / `libjiagu_art.so`
- Java 层特征:
 - 代理 Application 包名：`com.qihoo.util`。
 - 启动类中可能包含 `com.stub.StubApp`。
- `assets` 目录特征:
 - `libjiagu.so` (是的，有时也会放在 assets 里)
 - `.jiagu` 后缀的加密文件。

腾讯乐固 (Tencent Legu)

腾讯乐固通常与 Buggy SDK 一起出现，特征明显。

- SO 库特征:
 - `liblegu.so`
 - `libshella-xxxx.so` (xxxx 是版本号)
- Java 层特征:
 - 代理 Application 包名：`com.tencent.buggy.legu`。

- **assets** 目录特征:
- `legu_data.so`
- `tosversion` 文件
- 其他:
 - DEX 文件头通常被修改为 `legu`。

网易易盾 (Netease Yidun)

网易易盾是近年来兴起的一款强大加固，特征也比较独特。

- SO 库特征:
- `libnesec.so` (最核心的特征)
- Java 层特征:
- 代理 Application 包名: `com.netease.nis.wrapper`。
- **assets** 目录特征:
- `nesecc.dat`
- `classes.dex.ys` (加密的主 DEX)
- `xxx.dat` 格式的加密 DEX 文件。

爱加密 (ljiami)

爱加密也是一款常见的加固产品。

- SO 库特征:
- `libexec.so`
- `libexecmain.so`
- `libljiami.so`

- Java 层特征:
 - 代理 Application 包名: `com.ijiami.client.protect`。
 - `assets` 目录特征:
 - `ijiami.dat`
 - `ijm_lib` 目录
-

快速识别摘要表

加固厂商	核心 SO 特征	Java 包名/类名特征	assets 目录特征
梆梆安全	<code>libSecShell.so</code>	<code>com.bangcle.protect</code>	<code>bangcle_classes.jar</code>
360加固	<code>libjiagu.so</code> , <code>libprotectClass.so</code>	<code>com.qihoo.util</code>	<code>.jiagu</code> 文件
腾讯乐固	<code>liblegu.so</code>	<code>com.tencent.bugly.legu</code>	<code>legu_data.so</code>
网易易盾	<code>libnesec.so</code>	<code>com.netease.nis.wrapper</code>	<code>nesec.dat</code> , <code>classes.dex.ys</code>
爱加密	<code>libexec.so</code> , <code>libijiami.so</code>	<code>com.ijiami.client.protect</code>	<code>ijiami.dat</code>

[R10] 设备指纹与绕过

R10: 设备指纹技术深度解析与绕过策略

设备指纹 (Device Fingerprinting) 是指通过采集设备的软硬件特征，生成一个能够唯一标识该设备的、具有高熵值和稳定性的 ID 的过程。在当今的互联网服务中，它已成为反欺诈、反机器人、用户行为追踪和安全风控的基石技术。

绕过设备指纹并非简单地修改一两个参数，而是要创造一个完整的、逻辑自洽的、可信的虚拟设备“画像”。本指南将系统性地拆解主流的指纹采集维度，并探讨与之对应的核心绕过技术。

设备指纹的工作原理

指纹生成算法

设备指纹的生成并非简单地将所有采集到的信息拼接在一起，而是通过复杂的算法处理，确保生成的指纹具有唯一性、稳定性和不可逆性。

基本流程

对采集到的原始数据进行预处理：

- 格式统一：将不同格式的数据转换为标准格式（如MAC地址统一为小写、去除分隔符）
- 缺失值处理：对无法获取的字段使用默认值或特殊标记
- 权重分配：根据稳定性和唯一性给不同维度分配权重

特征组合 - 将处理后的数据按照预定规则组合：

```

# 概念代码
fingerprint_input = {
    'hardware': {
        'android_id': 'abc123',
        'imei': '867530900000000',
        'mac': '00:11:22:33:44:55'
    },
    'software': {
        'model': 'Pixel 6',
        'sdk': 33,
        'fingerprint': 'google/raven/raven:...'
    },
    'environment': {
        'screen': '1080x2400',
        'dpi': 420,
        'timezone': 'Asia/Shanghai'
    }
}

import hashlib
import json

def generate_fingerprint(data):
    # 数据转为JSON字符串（确保顺序一致）
    json_str = json.dumps(data, sort_keys=True)
    # 计算SHA256
    fingerprint = hashlib.sha256(json_str.encode()).hexdigest()
    return fingerprint

```

算法	输出长度	特点	适用场景
MD5	128位	速度快，但安全性低	低安全要求场景
SHA-256	256位	安全性高，计算稍慢	金融、高安全场景
MurmurHash	可变	速度极快，适合非加密	大规模数据处理
xxHash	可变	性能优异	实时计算场景

高级技术

1. 模糊Hash (Fuzzy Hashing)

允许设备指纹在细微变化时仍能匹配。使用 SimHash、MinHash 等算法：

```
def simhash(features, hash_bits=64):
    """
    将特征向量转换为 SimHash 值
    相似特征会产生相似的Hash值
    """
    v = [0] * hash_bits

    for feature, weight in features.items():
        h = hash(feature)
        for i in range(hash_bits):
            if h & (1 << i):
                v[i] += weight
            else:
                v[i] -= weight

    fingerprint = 0
    for i in range(hash_bits):
        if v[i] >= 0:
            fingerprint |= (1 << i)

    return fingerprint
```

2. 分层指纹

- 一级指纹（硬件指纹）：基于IMEI、Android ID等硬件ID
- 二级指纹（系统指纹）：基于系统版本、设备型号等
- 三级指纹（环境指纹）：基于网络、行为等临时特征

```
def generate_tiered_fingerprint(data):
    # 一级指纹: 硬件ID
    tier1 = hashlib.sha256(
        f"{{data['imei']}}|{{data['android_id']}}".encode()
    ).hexdigest()

    # 二级指纹: 系统特征
    tier2 = hashlib.sha256(
        f"{{tier1}}|{{data['model']}}|{{data['sdk']}}".encode()
    ).hexdigest()

    # 三级指纹: 完整特征
    tier3 = hashlib.sha256(
        json.dumps(data, sort_keys=True).encode()
    ).hexdigest()

    return {
        'strong': tier1,
        'medium': tier2,
        'weak': tier3
    }
```

3. 机器学习特征提取

- 使用 PCA（主成分分析）提取关键特征
- 使用聚类算法识别异常设备
- 使用深度学习模型生成设备嵌入向量（Embedding）

熵值与稳定性

好的设备指纹需要在唯一性（高熵值）和稳定性之间找到平衡。

熵值计算

熵值衡量一个特征的信息量和区分能力：

```

import math
from collections import Counter

def calculate_entropy(values):
    """
    计算一个特征的香农熵
    熵值越高，说明该特征区分能力越强
    """
    total = len(values)
    counter = Counter(values)

    entropy = 0
    for count in counter.values():
        p = count / total
        entropy -= p * math.log2(p)

    return entropy

# 示例
android_ids = ['id1', 'id2', 'id3', ...] # 采集的数据
entropy = calculate_entropy(android_ids)
print(f"Android ID 熵值: {entropy} bits")

```

特征	熵值	唯一性	稳定性
Android ID	60+ bits	高	中（恢复出厂会变）
MAC 地址	48 bits	高	中（越来越难获取）
设备型号	8-10 bits	低	高
屏幕分辨率	6-8 bits	低	高
传感器列表	15-20 bits	中	高

稳定性评估

稳定性指的是设备在不同时间点、不同环境下生成的指纹一致性：

```
def stability_score(fingerprints):
    """
    评估同一设备在不同时间生成的指纹稳定性
    fingerprints: 同一设备多次生成的指纹列表
    """
    if len(fingerprints) < 2:
        return 1.0

    # 计算指纹之间相似度
    base = fingerprints[0]
    similarities = []

    for fp in fingerprints[1:]:
        # Hamming 距离
        diff = sum(c1 != c2 for c1, c2 in zip(base, fp))
        similarity = 1 - (diff / len(base))
        similarities.append(similarity)

    return sum(similarities) / len(similarities)

def quality_score(entropy, stability, coverage):
    """
    计算指纹方案质量分数
    entropy: 熵值 (0-100)
    stability: 稳定性 (0-1)
    coverage: 设备覆盖率 (0-1)
    """
    # 加权计算
    score = (
        entropy * 0.4 +          # 唯一性权重 40%
        stability * 50 * 0.4 +    # 稳定性权重 40%
        coverage * 100 * 0.2      # 覆盖率权重 20%
    )
    return score
```

指纹更新策略

1. 主动更新触发条件

- 设备硬件变更（换SIM卡、重置设备）
- App版本升级（指纹算法更新）
- 定期刷新（如每30天）

2. 被动更新触发条件

- 检测到指纹冲突（多个设备具有相同指纹）

- 检测到异常行为（疑似改机）
- 服务端要求强制更新

更新策略实现

```
class FingerprintManager:  
    def should_update(self, old_fp, new_data):  
        """  
        检查是否需要更新指纹  
        """  
        # 计算新指纹  
        new_fp = self.generate_fingerprint(new_data)  
  
        # 1. 关键字段变更  
        critical_changed = self._check_critical_fields(old_fp, new_data)  
        if critical_changed:  
            return True, "Critical field changed"  
  
        # 2. 相似度过低  
        similarity = self._calculate_similarity(old_fp, new_fp)  
        if similarity < 0.7:  
            return True, "Low similarity"  
  
        # 3. 时间过期  
        if self._is_expired(old_fp):  
            return True, "Expired"  
  
        return False, "No update needed"  
  
    def update_fingerprint(self, device_id, new_fp, reason):  
        """  
        更新指纹时保留历史记录  
        """  
        history = {  
            'device_id': device_id,  
            'old_fingerprint': self.current_fp,  
            'new_fingerprint': new_fp,  
            'update_reason': reason,  
            'timestamp': time.time()  
        }  
        self._save_history(history)  
        self.current_fp = new_fp
```

主流设备指纹采集维度

硬件层标识符

这些是传统的、权限较高的设备 ID。

标识符	获取方式 (Java API)	特点
Android ID	<code>Settings.Secure.getString(resolver, "android_id")</code>	Android 8.0 以上，对每个 App 和用户都不同。恢复出厂设置会改变。
IMEI/ MEID	<code>TelephonyManager.getImei()</code>	手机的唯一身份码。需要 <code>READ_PHONE_STATE</code> 权限，且越来越难获取。
IMSI	<code>TelephonyManager.getSubscriberId()</code>	SIM 卡的唯一身份码。同样需要高权限。
MAC 地址	<code>WifiInfo.getMacAddress()</code>	Android 6.0 以后，App 获取到的通常是一个固定的假值 <code>02:00:00:00:00:00</code> 。

系统与软件特征

这是指纹库的主体，信息量大，获取成本低。

Build 属性：通过 `android.os.Build` 类或直接读取 `/system/build.prop` 文件获取。

- `Build.MODEL`：设备型号 (e.g., "Pixel 6")
- `Build.BRAND`：品牌 (e.g., "Google")
- `Build.MANUFACTURER`：制造商 (e.g., "Google")
- `Build.VERSION.SDK_INT`：SDK 版本号 (e.g., 33)
- `Build.FINGERPRINT`：系统构建指纹，信息量巨大。

系统设置:

- 屏幕分辨率、DPI (`DisplayMetrics`)
- 系统语言、时区、默认字体列表。

软件环境:

- 已安装应用列表 (`PackageManager.getInstalledPackages`)
- 特定 App (如输入法) 的版本

硬件特性指纹

利用硬件的细微物理差异来创建指纹。

- 传感器数据: 读取加速度计、陀螺仪等传感器的校准数据或在特定操作下的读数。不同批次的传感器存在物理差异。
- CPU/GPU 信息:
 - 读取 `/proc/cpuinfo` 获取 CPU 型号、核心数、特性等。
 - 通过 OpenGL/WebGL API 查询 GPU 供应商、渲染器信息，甚至可以执行一个标准渲染任务，将渲染结果的 Hash 作为指纹。
- 摄像头参数: `CameraCharacteristics` 中包含的详细参数。

通过 SVC (系统调用) 获取信息

这是一种高级的反 Hook 技术，常见于加固方案中。其核心思想是绕过所有上层 API 和 libc 函数，通过 `SVC` 指令直接发起系统调用 (`syscall`) 来获取信息或执行操作。

原理: `SVC` 是 ARM 处理器的一条指令，它会触发一个软件中断，使 CPU 从用户态 (User Mode) 切换到管理态 (Supervisor Mode)，从而执行内核代码。这是所有系统调用的基础。加固厂商在 SO 文件中直接嵌入 `SVC` 指令，可以不经过 `libc.so` 中的 `read`, `open`, `ioctl` 等函数，直接调用内核中对应的功能。

应用场景:

- 绕过 API Hook: 这是其最主要的目的。由于 Frida、Xposed 等框架主要 Hook 的是 App 进程空间中的函数（Java API 或 Native API），`SVC` 指令直接与内核交互，使得这些上层 Hook 完全失效。
- 读取敏感文件: 直接使用 `open` / `read` 的系统调用号来读取 `/proc/self/maps`，`/proc/cpuinfo` 等文件，以检测环境或收集指纹。
- 执行反调试: 使用 `ptrace` 的系统调用号来执行反调试检查。

分析与识别:

- 静态分析: 在 IDA 等反汇编工具中，直接搜索 `SVC` 指令。如果一个 SO 文件中含有大量 `SVC` 指令，且其上下文逻辑复杂，则极有可能使用了该技术。
- 动态分析: Hook 系统调用需要更底层的工具。Frida 的 `Stalker` 可以用来跟踪指令级的执行流程，从而捕捉到 `SVC` 的调用。

网络环境指纹

- IP 地址: 最基础的维度，结合地理位置库可以判断用户位置。
- 网络信息: 运营商名称 (`TelephonyManager.getNetworkOperatorName`)、Wi-Fi BSSID/SSID。
- TLS/JA3 指纹: 在建立 TLS 连接时，客户端 `Client Hello` 包的特征可以构成一个稳定的指纹，用于识别特定的网络库和版本。

行为特征指纹

行为特征是一种动态指纹，基于用户的操作模式和设备使用习惯。

采集维度

1. 触摸行为

```
view.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        // 采集触摸压力
        float pressure = event.getPressure();
        // 采集触摸面积
        float size = event.getSize();
        // 采集触摸坐标和时间戳
        long timestamp = event.getEventTime();
        float x = event.getX();
        float y = event.getY();

        // 构建触摸特征向量
        TouchFeature feature = new TouchFeature(pressure, size, timestamp, x, y);
        return false;
    }
});
```

2. 传感器行为

```
SensorManager sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

sensorManager.registerListener(new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
        float x = event.values[0];
        float y = event.values[1];
        float z = event.values[2];

        // 分析持握姿态、步态特征等
        analyzeMotionPattern(x, y, z);
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {}
}, accelerometer, SensorManager.SENSOR_DELAY_NORMAL);
```

3. 应用使用模式

- App 启动时间分布
- 常用 App 列表及使用频率
- 前后台切换模式
- 应用安装/卸载习惯

4. 网络行为

- 访问时间模式（工作日 vs 周末，白天 vs 晚上）
- 请求频率和间隔
- 网络切换习惯（WiFi ↔ 4G/5G）
- 常用地理位置

行为指纹生成

```
class BehaviorFingerprint:  
    def __init__(self):  
        self.touch_features = []  
        self.sensor_features = []  
        self.app_usage = {}  
        self.network_pattern = {}  
  
    def extract_touch_signature(self, touch_events):  
        """从触摸事件提取用户签名"""  
        pressures = [e.pressure for e in touch_events]  
        velocities = self._calculate_velocities(touch_events)  
  
        signature = {  
            'avg_pressure': np.mean(pressures),  
            'std_pressure': np.std(pressures),  
            'avg_velocity': np.mean(velocities),  
            'touch_rhythm': self._analyze_rhythm(touch_events)  
        }  
        return signature  
  
    def generate_behavior_fingerprint(self):  
        """生成综合行为指纹"""  
        touch_sig = self.extract_touch_signature(self.touch_features)  
        motion_sig = self.extract_motion_signature(self.sensor_features)  
        usage_sig = self.extract_usage_signature(self.app_usage)  
  
        # 组合为行为特征向量  
        behavior_vector = {  
            'touch': touch_sig,  
            'motion': motion_sig,  
            'usage': usage_sig,  
            'network': self.network_pattern  
        }  
  
        return hashlib.sha256(  
            json.dumps(behavior_vector, sort_keys=True).encode()  
        ).hexdigest()
```

核心绕过技术与策略

Hook 技术 (Frida/Xposed)

核心思路: 识别 -> Hook -> 伪造

1. 识别: 定位 App 获取关键指纹信息的代码位置 (Java API 或 JNI 函数)。
2. Hook: 使用 Frida 或 Xposed 拦截这些函数的调用。
3. 伪造: 在函数返回前, 用一套预设的、自洽的假数据替换真实返回值。

Frida 概念脚本 (伪造 Build.MODEL):

```
Java.perform(function () {
    var Build = Java.use("android.os.Build");
    Build.MODEL.value = "Pixel 4"; // 修改MODEL 字段

    var String = Java.use("java.lang.String");
    var TelephonyManager = Java.use("android.telephony.TelephonyManager");
    TelephonyManager.getDeviceId.overload().implementation = function () {
        console.log("Hooked getDeviceId(). Returning a fake IMEI.");
        return String.$new("867530900000000"); // 返回伪造 IMEI
    };
});
```

深度设备修改 ("改机")

需要 Root 权限, 直接修改 `/system/build.prop` 等系统级文件, 或通过内核模块修改系统调用的返回值。

- 优点: 无法通过应用层的检测手段识破, 因为 App 获取到的就是系统层返回的"真实"数据。
- 缺点: 技术门槛高, 工作量巨大。

环境虚拟化与容器技术

虚拟化和容器技术是规模化设备指纹绕过的核心基础设施, 能够在单台物理机上运行数百个独立的 Android 实例。

Android 虚拟化技术栈

1. 基于 QEMU 的完整虚拟化

Android 官方模拟器 (AVD) 基于 QEMU 实现：

```
# 启动 AVD 模拟器
emulator -avd Pixel_6_API_33 \
    -no-snapshot \
    -wipe-data \
    -gpu swiftshader_indirect
```

主要问题：

```
# 容易被检测的特征
getprop ro.hardware          # 返回 "goldfish" 或 "ranchu"
getprop ro.product.model      # 返回 "android SDK built for x86"
getprop ro.build.fingerprint # 包含 "通用" 字样

# 缺失传感器
pm list features | grep sensor # 大量传感器缺失
```

2. 基于容器的方案 (Docker/LXC)

容器技术提供更轻量的隔离：

```
# Dockerfile for Android container
FROM ubuntu:20.04

# 安装 Android 环境
RUN apt-get update && apt-get install -y \
    openjdk-11-jdk \
    android-sdk \
    adb \
    fastboot

# 配置 Android 环境
ENV ANDROID_HOME=/opt/android-sdk
ENV PATH=$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools

# 运行 ADB 服务
CMD ["adb", "-a", "nodaemon", "server"]
```

3. 专业容器方案：Redroid

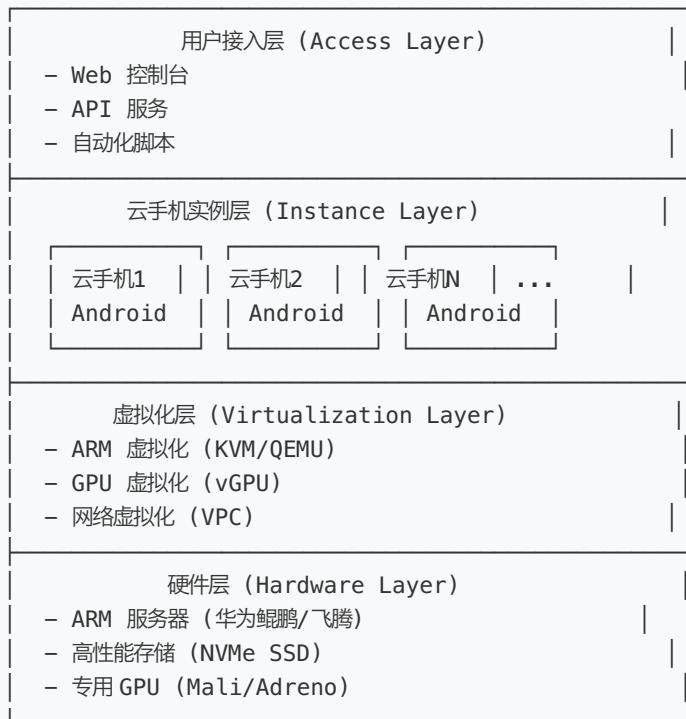
Redroid 是一个基于 Docker 的 Android 容器方案：

```
# 运行 Redroid 容器
docker run -d \
    --name redroid \
    --privileged \
    -v ~/data:/data \
    -p 5555:5555 \
    redroid/redroid:11.0.0-latest

# 连接到 Redroid
adb connect localhost:5555
adb shell
```

云手机技术详解

云手机架构



云手机平台对比

平台	架构	密度	性能	成本	适用场景
华为云手机	ARM 服务器+KVM	中	高	高	企业级应用
红手指	ARM 容器	高	中	中	自动化、挂机
多多云手机	x86+容器	高	低	低	批量注册、养号
AWS Device Farm	真机	低	极高	极高	测试、兼容性验证

构建一致性的"设备画像"

```
class DeviceFingerprintManager:  
    def __init__(self, device_pool_db):  
        self.db = device_pool_db  
        self.used_fingerprints = set()  
  
    def get_unused_fingerprint(self):  
        """从设备池中获取未使用的指纹"""  
        while True:  
            fp = self.db.get_random_fingerprint()  
            fp_hash = hashlib.md5(json.dumps(fp).encode()).hexdigest()  
  
            if fp_hash not in self.used_fingerprints:  
                self.used_fingerprints.add(fp_hash)  
                return fp  
  
    def apply_fingerprint(self, adb_device, fingerprint):  
        """将指纹应用到设备"""  
        # 修改系统属性  
        for key, value in fingerprint['build_props'].items():  
            adb_device.shell(f"setprop {key} {value}")  
  
        # 安装预设 App  
        for apk in fingerprint['apps']:  
            adb_device.install(apk)  
  
        # 设置位置  
        adb_device.shell(f"settings put secure location_mode 3")  
        adb_device.shell(  
            f"am startservice -a com.example.fakelocation "  
            f"--es lat {fingerprint['location']['lat']} "  
            f"--es lng {fingerprint['location']['lng']}"  
        )  
  
    def rotate_fingerprint(self, adb_device, interval_hours=24):  
        """定期轮换设备指纹"""  
        while True:  
            new_fp = self.get_unused_fingerprint()  
            self.apply_fingerprint(adb_device, new_fp)  
  
            # 记录使用历史  
            self.db.log_usage(adb_device.serial, new_fp, timestamp=time.time())  
  
            time.sleep(interval_hours * 3600)
```

商业化产品与服务

国内主流设备指纹服务商

1. 顶象科技 (DingXiang)

产品: 顶象设备指纹 (Device Fingerprint)

技术特点:

- 采集 200+ 设备特征维度
- 支持 Android、iOS、Web、小程序
- 99.9%+ 设备唯一性识别率
- 设备指纹有效期 90 天+
- 支持私有化部署

定价: 按 API 调用次数计费, 企业版约 0.005-0.01 元/次

2. 同盾科技 (Tongdun)

产品: 同盾设备指纹

技术特点:

- 结合 AI 和大数据分析
- 设备行为画像
- 实时风险决策
- 覆盖金融、电商、O2O 等场景

API 示例:

```
import requests
import time

def tongdun_device_risk(device_id, event_type):
    """调用同盾设备风险评估 API"""
    url = "https://api.tongdun.cn/riskService"

    params = {
        "partner_code": "your_partner_code",
        "device_id": device_id,
        "event_type": event_type, # 如: login, register, pay
        "timestamp": int(time.time() * 1000)
    }

    # 添加签名
    params["sign"] = generate_sign(params)

    response = requests.post(url, json=params)
    result = response.json()

    return {
        "risk_score": result["final_score"], # 风险分数 0-100
        "risk_level": result["risk_level"], # high/medium/low
        "device_labels": result["labels"] # 设备标签
    }
```

3. 数美科技 (ISHUMEI)

技术特点：

- 专注于内容安全和业务安全
- 设备指纹+行为分析
- 实时黑产设备库
- 支持多场景风控

应用场景：

- 羊毛党识别
- 虚假注册拦截
- 刷单检测
- 恶意爬虫识别

4. 网易易盾 (NetEase YiDun)

产品: 易盾设备指纹

技术特点:

- 网易内部风控技术外化
- 游戏、社交场景优化
- 设备唯一性识别
- 设备环境检测 (Root、模拟器、Hook 框架)

SDK 集成示例 (Android) :

```
// 初始化
NECaptcha.getInstance()
    .init(context, "your_business_id", new NECaptchaListener() {
        @Override
        public void onReady() {
            // 获取设备指纹
            String deviceId = NEDeviceRisk.getDeviceId();
        }
    });
}

// 获取设备风险信息
NEDeviceRisk.check(context, new NEDeviceRiskCallback() {
    @Override
    public void onResult(NEDeviceRiskResult result) {
        int riskLevel = result.getRiskLevel(); // 0-4 级风险
        boolean isEmulator = result.isEmulator();
        boolean isRooted = result.isRoot();
        boolean isHooked = result.isHook();
    }
});
```

国际知名产品

1. FingerprintJS

类型: 开源 + 商业版

特点：

- 主要用于 Web 浏览器指纹
- 开源版本基础功能免费
- Pro 版提供 99.5% 准确率

使用示例：

```
import FingerprintJS from "@fingerprintjs/fingerprintjs";

// 初始化 agent
const fpPromise = FingerprintJS.load();

// 获取访客标识
fpPromise
  .then((fp) => fp.get())
  .then((result) => {
    // 访客标识
    const visitorId = result.visitorId;
    console.log(visitorId);

    // 所有组件 (浏览器特征)
    console.log(result.components);
  });
}
```

2. DeviceAtlas

特点：

- 包含数万种设备型号
 - 主要用于移动广告和分析
 - 支持云端 API 和本地部署
-

开源工具与框架

设备指纹采集框架

1. FingerprintJS

- GitHub: <https://github.com/fingerprintjs/fingerprintjs>
- 轻量级浏览器指纹库
- 纯 JavaScript 实现
- 采集 Canvas、WebGL、Audio 等特征

2. android-device-names

- 支持 12000+ 设备型号
- 可以根据 Build.MODEL 获取市场化设备名称

```
DeviceName.with(context).request(new DeviceName.Callback() {  
    @Override  
    public void onFinished(DeviceName.DeviceInfo info, Exception error) {  
        String manufacturer = info.manufacturer; // "Samsung"  
        String marketName = info.marketName; // "Galaxy S21"  
        String model = info.model; // "SM-G991B"  
        String codename = info.codename; // "o1s"  
    }  
});
```

反指纹工具

1. Magisk 模块

```
# 修改设备指纹  
props ro.build.fingerprint "google/raven/raven:12/..."  
props ro.product.model "Pixel 6"  
  
# 应用修改  
props ro.build.product "raven"
```

2. Xposed 模块

XPrivacyLua

- GitHub: <https://github.com/M66B/XPrivacyLua>
- 细粒度权限控制
- API 返回值 Hook
- 设备信息伪造

3. Frida 脚本库

常用的设备信息 Hook 脚本:

```
Java.perform(function() {
    // 拦截 Build 类所有字段
    var Build = Java.use("android.os.Build");
    Build.BRAND.value = "google";
    Build.MODEL.value = "Pixel 6";
    Build.DEVICE.value = "raven";
    Build.PRODUCT.value = "raven";
    Build.MANUFACTURER.value = "Google";

    // Hook Settings.Secure
    var Settings = Java.use("android.provider.Settings$Secure");
    Settings.getString.overload(
        "android.content.ContentResolver",
        "java.lang.String"
    ).implementation = function(resolver, name) {
        if (name == "android_id") {
            return "fake_android_id_12345678";
        }
        return this.getString(resolver, name);
    };

    // Hook TelephonyManager
    var TelephonyManager = Java.use("android.telephony.TelephonyManager");
    TelephonyManager.getDeviceId.overload().implementation = function() {
        return "fake_imei_123456789012345";
    };
});
```

对抗与挑战

Hook 框架检测

- 检查 `/proc/self/maps` 中是否加载了 `frida-agent.so` 或 `XposedBridge.jar`
- 检测 Frida 的默认端口 `27042`
- 通过 `try-catch` 执行一个会因 Xposed 修改而改变行为的函数，判断是否抛出异常

服务端交叉验证

这是设备指纹技术最强大的地方。后端服务会将客户端上传的几百个维度的指纹数据进行交叉比对。一个 `IMEI` 显示是三星设备，但 `Build.FINGERPRINT` 却属于小米，这种矛盾会立刻导致该设备被标记为高风险。

交叉验证规则示例：

```
class FingerprintValidator:  
    def __init__(self):  
        self.device_database = self.load_device_db()  
        self.inconsistency_rules = self.load_rules()  
  
    def validate_fingerprint(self, fingerprint):  
        """验证设备指纹一致性"""  
        issues = []  
  
        # 规则1: 品牌与型号匹配  
        if not self.check_brand_model_match(  
            fingerprint['brand'],  
            fingerprint['model'])  
        ):  
            issues.append({  
                'type': 'brand_model_mismatch',  
                'severity': 'high',  
                'message': f"Brand {fingerprint['brand']} does not match model  
{fingerprint['model']}"  
            })  
  
        # 规则2: 屏幕分辨率与型号匹配  
        expected_resolution =  
            self.device_database.get_resolution(fingerprint['model'])  
        if fingerprint['screen_resolution'] != expected_resolution:  
            issues.append({  
                'type': 'resolution_mismatch',  
                'severity': 'medium',  
                'message': f"Unexpected resolution for {fingerprint['model']}"  
            })  
  
        # 规则3: 传感器列表完整性  
        expected_sensors = self.device_database.get_sensors(fingerprint['model'])  
        if len(fingerprint['sensors']) < len(expected_sensors) * 0.8:  
            issues.append({  
                'type': 'sensor_missing',  
                'severity': 'high',  
                'message': 'Too few sensors for this device model'  
            })  
  
        return {  
            'is_valid': len(issues) == 0,  
            'risk_score': self.calculate_risk_score(issues),  
            'issues': issues  
        }
```

机器学习检测

使用标注数据训练模型，识别真实设备 vs 伪造设备：

```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.neural_network import MLPClassifier
import numpy as np

class DeviceAuthenticityClassifier:
    def __init__(self):
        self.models = {
            'random_forest': RandomForestClassifier(n_estimators=200, max_depth=15),
            'gradient_boosting': GradientBoostingClassifier(n_estimators=100),
            'neural_network': MLPClassifier(hidden_layer_sizes=(128, 64, 32))
        }
        self.feature_extractor = FeatureExtractor()

    def train(self, X_train, y_train):
        """
        训练分类器
        X_train: 设备指纹特征
        y_train: 标签 (0=伪造, 1=真实)
        """
        for name, model in self.models.items():
            print(f"Training {name}...")
            model.fit(X_train, y_train)

    def predict(self, fingerprint):
        """
        预测设备真实性
        features = self.feature_extractor.extract(fingerprint)

        # 集成投票
        votes = []
        probabilities = []

        for name, model in self.models.items():
            pred = model.predict([features])[0]
            prob = model.predict_proba([features])[0]
            votes.append(pred)
            probabilities.append(prob[1]) # 真实设备概率

        # 加权平均
        avg_probability = np.mean(probabilities)

        return {
            'is_genuine': avg_probability > 0.5,
            'confidence': avg_probability,
            'votes': dict(zip(self.models.keys(), votes))
        }

```

实战案例分析

案例 1：某电商平台设备指纹分析

背景：某大型电商平台面临大量刷单、虚假评论和薅羊毛行为，需要通过设备指纹识别恶意用户。

技术方案：

1. 指纹采集

```
public class EcommerceFingerprintCollector {  
    public DeviceFingerprint collect(Context context) {  
        DeviceFingerprint fp = new DeviceFingerprint();  
  
        // 基本硬件信息  
        fp.setAndroidId(getAndroidId(context));  
        fp.setModel(Build.MODEL);  
        fp.setBrand(Build.BRAND);  
  
        // 网络信息  
        fp.setIpAddress(getIPAddress());  
        fp.setMacAddress(getMacAddress(context));  
  
        // App 信息  
        fp.setInstalledApps(getInstalledApps(context));  
  
        // 行为特征  
        fp.setTouchPressure(collectTouchPressure());  
        fp.setTypingSpeed(collectTypingSpeed());  
  
        // 环境检测  
        fp.setIsRooted(checkRootStatus());  
        fp.setIsEmulator(checkEmulatorStatus());  
        fp.setIsHooked(checkHookStatus());  
  
        return fp;  
    }  
  
    private boolean checkHookStatus() {  
        // 检测 Frida  
        if (checkFridaPort()) return true;  
        if (checkFridaLibraries()) return true;  
  
        // 检测 Xposed  
        if (checkXposedEnvironment()) return true;  
  
        return false;  
    }  
}
```

2. 风控决策

```
class EcommerceRiskEngine:  
    def __init__(self):  
        self.fingerprint_db = FingerprintDatabase()  
        self.ml_detector = DeviceAuthenticityClassifier()  
        self.behavior_analyzer = BehaviorSequenceDetector()  
  
    def check_order_risk(self, user_id, device_fp, order_info):  
        """订单风险评估"""  
        risk_factors = []  
        risk_score = 0  
  
        # 1. 设备指纹检查  
        device_risk = self.check_device_fingerprint(device_fp)  
        if device_risk['is_suspicious']:  
            risk_factors.append(device_risk)  
            risk_score += 30  
  
        # 2. 设备关联分析  
        related_devices = self.fingerprint_db.find_related_devices(device_fp)  
        if len(related_devices) > 10:  
            risk_factors.append({  
                'type': 'device_cluster',  
                'message': f'Device associated with {len(related_devices)} other  
devices'  
            })  
            risk_score += 25  
  
        # 3. 用户行为分析  
        user_behavior = self.fingerprint_db.get_user_behavior(user_id)  
        if self.is_bot_behavior(user_behavior):  
            risk_factors.append({'type': 'bot_behavior'})  
            risk_score += 35  
  
        # 决策  
        if risk_score >= 60:  
            action = 'reject'  
        elif risk_score >= 40:  
            action = 'manual_review'  
        else:  
            action = 'approve'  
  
        return {  
            'action': action,  
            'risk_score': risk_score,  
            'risk_factors': risk_factors  
        }
```

案例 2：金融 App 风控绕过

背景：某金融 App 使用顶象设备指纹进行风控，攻击者尝试绕过进行批量注册和薅羊毛。

App 保护措施：

1. 集成顶象 SDK 采集设备指纹
2. SO 层加固（360 加固）
3. 检测 Root、模拟器、Hook 框架
4. 网络请求签名验证

分析过程：

Step 1: 设备指纹 SDK 定位

```
# 反编译 APK  
apktool d app.apk  
  
# 搜索设备指纹相关代码  
grep -r "getDeviceId" .  
grep -r "fingerprint" .  
  
# 找到 SDK 包名  
# com.dingxiang.sdk.fingerprint
```

Step 2: Frida Hook 分析

```
Java.perform(function() {
    // Hook 顶象 SDK
    var DXFingerprint = Java.use("com.dingxiang.sdk.fingerprint.DXFingerprint");

    DXFingerprint.getDeviceId.implementation = function() {
        console.log("[*] getDeviceId() called");

        // 返回伪造设备 ID
        var fakeDeviceId = "fake_dx_device_id_" +
            Math.random().toString(36).substring(7);
        console.log("[*] Returning fake device ID: " + fakeDeviceId);

        return fakeDeviceId;
    };

    // Hook 采集方法
    DXFingerprint.collect.implementation = function(context) {
        console.log("[*] collect() called");

        // 修改采集数据
        var result = this.collect(context);

        // 篡改指纹数据
        modifyFingerprintData(result);

        return result;
    };
});
```

教训：

1. 单纯的 Hook 不够：需要完整的环境伪装
2. 设备一致性至关重要：所有参数必须逻辑自洽
3. 行为模拟必不可少：纯技术绕过容易被行为分析识破
4. 成本与收益平衡：高质量绕过需要较高成本

总结

成功的绕过，本质上是一场关于“伪造一个天衣无缝的设备画像”的持久战。

[R11] 移动安全与反爬

R11: 移动端安全与风控技术

现代移动应用，特别是那些处理敏感用户数据或有价值业务逻辑的应用，通常会实现多层安全机制来防御逆向工程、篡改和自动化滥用（机器人）。这一领域通常被称为“RASP”（运行时应用自我保护）或“反机器人技术”。

本文档概述了常见的技术。

关键防御类型

1. 反调试与反分析

这些技术旨在检测并阻止像调试器和插桩框架这样的分析工具。

- 调试器检测:
 - 检查 `/proc/self/status` 中的 `TracerPid`。一个非零值表示有调试器附加。
 - 使用 `ptrace(PTRACE_TRACEME, 0, NULL, NULL)` 并检查返回值是否小于 0。
 - 时间检查：测量执行一段代码所需的时间。如果附加了调试器（带有断点），时间将会有显著不同。
- Frida 检测:
 - 扫描内存中是否存在 `frida-agent` 或 `frida-gadget` 库。
 - 通过尝试连接其默认端口（27042）来检测 Frida 的服务器。
 - 检查 Frida 特有的痕迹，如命名管道（`frida-pipe`）或对 `REPL` 环境的修改。
- 模拟器/Root 检测:

- 检查已知的模拟器特有文件、属性（`ro.kernel.qemu`）或设备驱动。
- 检查是否存在 root 管理应用（如 SuperSU）或 `su` 二进制文件。

2. 反篡改

这些技术确保应用的代码和数据未被修改。

- 代码完整性校验：在运行时计算 `classes.dex` 文件或原生库（`.so`）的校验和或哈希值（例如 SHA-256），并与安全存储的已知良好值进行比较。
- 签名验证：在运行时，获取应用自身的签名并验证其是否与官方开发者签名匹配。这可以防止重打包。
- 内存完整性校验：定期扫描应用自身的内存，以确保关键函数未被 Frida 等工具钩住或修补。

3. 反机器人与业务逻辑保护

这些技术旨在防止自动化脚本滥用应用的 API 或功能（例如，创建垃圾账户、抓取数据）。

- 设备证明：使用像 Android 的 `SafetyNet` / `Play Integrity` API 这样的服务，从谷歌服务器获取一个加密证明，证明设备是真实的、非 root 的，并且正在运行官方应用。这非常难以绕过。
 - 请求签名：关键的 API 请求通常受“签名”保护。这个签名是根据请求参数、时间戳、随机数和一个密钥计算出的哈希值（例如 HMAC-SHA256）。密钥通常使用混淆等技术隐藏起来。逆向这个签名算法是分析师的一个常见目标。
 - 行为分析：在服务器端，分析 API 调用的时间和顺序，以建立真实用户的行为画像。机器人通常具有非常僵硬、非人类的时间模式，可以用来检测它们。
-

绕过策略

绕过这些防御需要结合静态和动态分析。

- 钩子与补丁: 使用 Frida 钩住负责这些检查的函数, 并强制它们返回一个"安全"的值。例如, 钩住读取 `TracerPid` 的函数, 使其总是返回 0。
- 自定义 Frida 版本: 为了对抗 Frida 检测, 你可以编译一个自定义名称和不同默认端口的 `frida-server` 版本。
- 静态分析: 反汇编应用以找到检查逻辑。一旦找到, 你通常可以将其"nop"掉 (用 `NOP` 指令 替换), 以永久禁用它。

脱壳技术

[R12] 脱壳概述

R12: 脱壳分析加固的 Android 应用

问题场景

你在逆向分析一个 App 时遇到了以下情况：

- ✗ JADX 打开 APK 后代码完全不可读，全是混淆的类名或空方法
- ✗ classes.dex 文件异常小（几十 KB），不符合应用实际规模
- ✗ 应用启动时检测到 Frida 并闪退，常规 Hook 无法生效
- ✗ AndroidManifest.xml 中的 Application 入口被替换成可疑的壳类名
- ✗ `assets` 或 `lib` 目录中存在加密文件，如 `.dat`、`.bin` 或奇怪命名的 `.so` 文件

这些都是应用被加固(加壳)的典型特征。加固技术通过加密 DEX 文件、抽取方法体、虚拟化指令等手段，让静态分析工具无法直接读取原始代码。本配方将教你如何识别、脱壳并恢复被加固的应用。

工具清单

必需工具

- [x] Frida - 动态插桩框架
- [x] frida-dexdump - 自动化 DEX dumper ([GitHub](#))
- [x] ADB - 设备通信工具
- [x] Root 权限设备 或模拟器 (必须)

可选工具

- FUPK3 - 针对特定壳的专用脱壳工具
- Youpk - 较新的脱壳工具
- PKid/ApkTool-Plus - 加固类型识别工具
- MT 管理器 - Android 端 APK 分析工具
- IDA Pro/Ghidra - Native 层分析 (SO 加固时需要)

前置条件

在开始前请确认：

- 设备已 Root 并安装 Frida Server 了解 DEX 文件基本结构 (至少知道 magic number 0x6465780A) 应用已安装并能正常启动 (即使有反调试) 磁盘空间充足 (脱壳可能产生大量文件)

解决方案

核心原理

"代码运行必解密"

无论加固技术多么复杂，加密后的代码最终都必须在内存中恢复成可执行的 DEX 格式，才能被 ART 执行。脱壳的核心思想是：在代码被解密后、执行前的那一刻，从内存中将其 dump 出来。

第1步: 识别加固类型 (5-10分钟)

不同代际的加固技术需要不同的脱壳策略, 先识别目标应用使用了什么加固技术。

方法 A: 使用工具快速识别

```
# 使用 PKid (ApkTool-Plus) 检测
# 下载: https://github.com/rover12421/ApkToolPlus
java -jar ApkToolPlus.jar -pkid target.apk

# 输出示例:
# [+] 检测到加固厂商: 腾讯乐固 (Tencent Legu)
# [+] 加固类型: 第二代壳 (方法抽取)
```

方法 B: 手动检测

```
# 1. 检查 AndroidManifest.xml 中的 Application 类
unzip -p target.apk AndroidManifest.xml | strings | grep -i "application"

# 可疑类名:
# com.tencent.StubShell.TxAppEntry (腾讯乐固)
# com.secneo.apkwrapper.ApplicationWrapper (梆梆 Security)
# com.baidu.protect.StubApplication (百度加固)

# 2. 检查 DEX 文件大小
unzip -l target.apk | grep classes.dex
# 如果 classes.dex < 100KB 且 App 功能复杂, 很可能加壳

# 3. 检查可疑文件
unzip -l target.apk | grep -E "\.dat|\.bin|ijm_lib|secdata"
# 这些文件通常包含加密的原始 DEX

# 4. 检查 lib 目录中的可疑 SO
unzip -l target.apk | grep "lib/.*\.so" | grep -E "(exec|vmp|protect)"
```

加固技术代际对照表

代际	时期	技术特点	典型厂商	识别特征	脱壳难度
第一代	2010-2015	整体 DEX 加密	早期爱加密、360	Application 入口被替换	简单
第二代	2015-2018	方法抽取(Stolen Code)	腾讯乐固、阿里聚安全	大量空方法、libexec.so	中等
第三代	2018-2021	指令虚拟化(VMP)	梆梆 VMP、顶象科技	自定义 VM 引擎、私有指令	困难
第四代	2021-至今	云端+多重保护	腾讯御安全、阿里云	云端下发代码、多层次加壳	极难

第 2 步: 选择脱壳策略 (5 分钟)

根据识别出的加固类型，选择合适的脱壳方法：

第一代壳 (整体加密)

策略: Hook ClassLoader, 在 DEX 加载时 dump 推荐工具: 手写 Frida 脚本或 frida-dexdump 成功率: 95%+

第二代壳 (方法抽取)

策略: Hook ArtMethod 的 invoke, 在方法首次调用时 dump CodeItem 推荐工具: FART 技术 (Frida ART Hook) + frida-dexdump 成功率: 80%+ (取决于代码覆盖率)

第三代壳 (虚拟化)

策略: Hook 虚拟机引擎, 获取指令流 + 映射表逆向 推荐工具: IDA Pro + 自定义脚本 成功率: 50% (需要深入分析虚拟机实现)

第四代壳 (云端)

策略: 网络抓包 + 内存扫描 + 多层 dump 推荐工具: mitmproxy + frida-dexdump + 自定义脚本 成功率: 30% (部分逻辑可能无法获取)

第 3 步: 执行脱壳 (10-60 分钟)

以下提供针对不同代际的脱壳脚本。

方法 A: 使用 frida-dexdump (通用, 推荐首选)

```
# 1. 启动 Frida Server
adb push frida-server /data/local/tmp/
adb shell "chmod 755 /data/local/tmp/frida-server"
adb shell "/data/local/tmp/frida-server &"

# 2. 安装 frida-dexdump
pip3 install frida-dexdump

# 3. 运行脱壳 (Spawn 模式, 在 App 启动时拦截)
python3 -m frida_dexdump -U -f com.target.app -o ./output

# 输出示例:
# [+] Hooking ClassLoader...
# [+] Dumped DEX: /output/com.target.app_1234567890.dex (5.2 MB)
# [+] Dumped DEX: /output/com.target.app_0987654321.dex (1.8 MB)
# [+] Total: 2 DEX files dumped

# 4. 将 DEX 文件拉取到本地
adb pull /data/data/com.target.app/files/*.dex ./dumped_dex/
```

参数说明:

- `-U`: 使用 USB 连接的设备
- `-f com.target.app`: Spawn 模式启动应用

-
- `-o ./output` : 输出目录
-

方法 B: 手写 Hook 脚本 (第一代壳)

适用于简单的整体加密壳。

```
// unpacker_gen1.js - 第一代壳通用脚本

Java.perform(function () {
    console.log("[+] 开始 Hook ClassLoader...");

    // 拦截 1: DexClassLoader (最常见)
    var DexClassLoader = Java.use("dalvik.system.DexClassLoader");
    DexClassLoader.$init.implementation = function (
        dexPath,
        optimizedDirectory,
        librarySearchPath,
        parent
    ) {
        console.log("[+] DexClassLoader 加载 DEX:");
        console.log("    路径: " + dexPath);

        // 尝试复制 DEX 文件
        if (dexPath && dexPath.indexOf("/data/data/") !== -1) {
            try {
                var File = Java.use("java.io.File");
                var FileInputStream = Java.use("java.io.FileInputStream");
                var FileOutputStream = Java.use("java.io.FileOutputStream");

                var srcFile = File.$new(dexPath);
                if (srcFile.exists()) {
                    var timestamp = Date.now();
                    var dstPath =
                        "/data/data/com.target.app/dumped_" + timestamp + ".dex";
                    var dstFile = File.$new(dstPath);

                    var fis = FileInputStream.$new(srcFile);
                    var fos = FileOutputStream.$new(dstFile);

                    var buffer = Java.array("byte", [1024]);
                    var len;
                    while ((len = fis.read(buffer)) > 0) {
                        fos.write(buffer, 0, len);
                    }

                    fis.close();
                    fos.close();

                    console.log("✓ [已导出] " + dstPath);
                }
            } catch (e) {
                console.log("[-] 复制失败: " + e);
            }
        }
    }

    return this.$init(dexPath, optimizedDirectory, librarySearchPath, parent);
});
```

```
// Hook 2: InMemoryDexClassLoader (内存加载)
try {
    var InMemoryDexClassLoader = Java.use(
        "dalvik.system.InMemoryDexClassLoader"
    );
    InMemoryDexClassLoader.$init.overload(
        "java.nio.ByteBuffer",
        "java.lang.ClassLoader"
    ).implementation = function (byteBuffer, classLoader) {
        console.log("[+] InMemoryDexClassLoader 加载内存 DEX");

        // 从 ByteBuffer 提取 DEX
        try {
            var remaining = byteBuffer.remaining();
            console.log("    DEX 大小: " + remaining + " bytes");

            // 获取字节数组
            var bytes = Java.array("byte", [remaining]);
            byteBuffer.get(bytes);

            // 写入文件
            var timestamp = Date.now();
            var dstPath = "/data/data/com.target.app/memory_" + timestamp + ".dex";
            var File = Java.use("java.io.File");
            var FileOutputStream = Java.use("java.io.FileOutputStream");

            var dstFile = File.$new(dstPath);
            var fos = FileOutputStream.$new(dstFile);
            fos.write(bytes);
            fos.close();

            console.log("✓ [已导出] " + dstPath);

            // 重置 ByteBuffer 位置
            byteBuffer.position(0);
        } catch (e) {
            console.log("[-] 导出失败: " + e);
        }
    };

    return this.$init(byteBuffer, classLoader);
};
} catch (e) {
    console.log("[-] InMemoryDexClassLoader 不存在 (Android < 8.0)");
}

console.log("[+] Hook 完成, 等待 DEX 加载...");  
});
```

使用方法:

```
# Spawn 模式 (推荐)
frida -U -f com.target.app -l unpacker_gen1.js --no-pause

# Attach 模式
frida -U com.target.app -l unpacker_gen1.js
```

方法 C: FART 技术 (第二代壳 - 方法抽取)

FART (Frida-ART-Hook) 是针对方法抽取壳的高级技术。

```
// unpacker_fart.js - FART (Frida-ART-Hook) 脚本

// 警告：此脚本需要深入理解 ART 内部机制，不同 Android 版本可能需要调整偏移

var artMethodInvokeAddr = null;

// 根据 Android 版本查找 ArtMethod::Invoke 符号
var symbols = [
    "_ZN3art9ArtMethod6InvokeEPNS_6ThreadEPjjPNS_6JValueEPKc", // Android 7.0+
    "_ZN3art9ArtMethod6InvokeEPNS_6ThreadEPjmPNS_6JValueEPKc", // Android 8.0+
];
for (var i = 0; i < symbols.length; i++) {
    artMethodInvokeAddr = Module.findExportByName("libart.so", symbols[i]);
    if (artMethodInvokeAddr) {
        console.log("[+] 找到 ArtMethod::Invoke: " + artMethodInvokeAddr);
        break;
    }
}

if (!artMethodInvokeAddr) {
    console.log("[-] 未找到 ArtMethod::Invoke, 无法继续");
} else {
    Interceptor.attach(artMethodInvokeAddr, {
        onEnter: function (args) {
            var artMethod = args[0];

            // 读取 ArtMethod 结构中的 CodeItem (偏移因版本而异)
            // 这里以 Android 7.0 为例，实际使用需要根据版本调整
            try {
                // 获取方法名 (通过 PrettyMethod)
                var prettyMethodAddr = Module.findExportByName(
                    "libart.so",
                    "_ZN3art9ArtMethod12PrettyMethodEv"
                );
                if (prettyMethodAddr) {
                    var prettyMethod = new NativeFunction(prettyMethodAddr, "pointer", [
                        "pointer",
                    ]);
                    var methodName = prettyMethod(artMethod).readCString();

                    // 只关注 App 自身方法，忽略系统方法
                    if (methodName && methodName.indexOf("com.target.app") !== -1) {
                        console.log("[+] 调用方法: " + methodName);

                        // 尝试获取 CodeItem
                        // 注意：CodeItem 偏移在不同版本中不同
                        // Android 7.0: offset 24
                        // Android 8.0+: offset 16
                        var codeItemOffset = 24; // 需要根据实际版本调整
                        var codeItemPtr = artMethod.add(codeItemOffset).readPointer();
                    }
                }
            } catch (e) {
                console.error("Error reading CodeItem: " + e.message);
            }
        }
    });
}
```

```
if (codeItemPtr && !codeItemPtr.isNull()) {
    // 读取 CodeItem 结构
    var registersSize = codeItemPtr.readU16();
    var insSize = codeItemPtr.add(2).readU16();
    var outsSize = codeItemPtr.add(4).readU16();
    var triesSize = codeItemPtr.add(6).readU16();
    var insnsSize = codeItemPtr.add(12).readU32();

    if (insnsSize > 0 && insnsSize < 100000) {
        console.log("    找到 CodeItem: insnsSize = " + insnsSize);

        // 导出字节码
        var insnsPtr = codeItemPtr.add(16);
        var codeData = Memory.readByteArray(insnsPtr, insnsSize * 2);

        // 保存到文件
        var safeMethodName = methodName.replace(/[^\u00a1-\u00d7\u00d8-\u00d9]/g, "_");
        var filename =
            "/data/data/com.target.app/code_" + safeMethodName + ".bin";
        var file = new File(filename, "wb");
        file.write(codeData);
        file.close();

        console.log("✓ [已导出 CodeItem] " + filename);
    }
}
}

}

} catch (e) {
    // 忽略读取错误
}

},
});

console.log("[+] FART Hook 已激活, 开始监控方法调用...");
```

注意事项:

- 此技术需要较深的 ART 内部知识
- 只能 dump 被调用过的方法, 未触发的方法无法恢复

方法 D: 内存扫描 (通用兜底方案)

当其他方法失效时, 可以定期扫描内存中的 DEX magic number。

```
// unpacker_memscan.js - 内存扫描脚本

function scanMemoryForDex() {
    console.log("[+] 开始扫描内存中的 DEX 文件...");

    var ranges = Process.enumerateRanges("r--"); // 仅扫描可读区域
    var found = 0;

    ranges.forEach(function (range) {
        try {
            // DEX magic: "dex\n" = 0x6465780A
            var pattern = "64 65 78 0a";

            Memory.scan(range.base, range.size, pattern, {
                onMatch: function (address, size) {
                    console.log("[+] 发现潜在 DEX: " + address);

                    try {
                        // 读取 DEX 文件大小 (偏移 32 字节处)
                        var dexSize = address.add(32).readU32();

                        // 合理性检查
                        if (dexSize > 0x1000 && dexSize < 50 * 1024 * 1024) {
                            console.log(
                                "    DEX Size: " + (dexSize / 1024).toFixed(2) + " KB"
                            );
                        }

                        // Dump DEX
                        var dexData = Memory.readByteArray(address, dexSize);
                        var filename =
                            "/data/data/com.target.app/memdump_" +
                            address.toString().replace("0x", "") +
                            ".dex";
                        var file = new File(filename, "wb");
                        file.write(dexData);
                        file.close();

                        console.log("✓ [Dumped] " + filename);
                        found++;
                    }
                }
            } catch (e) {
                // 读取失败, 跳过
            }
        },
        onComplete: function () {},
    });
} catch (e) {
    // 忽略无法访问的内存区域
}

console.log("[+] 扫描完成, 找到 " + found + " 个 DEX 文件");
```

```
}

// 每 5 秒扫描一次
setInterval(function () {
    scanMemoryForDex();
}, 5000);

console.log("[+] 内存扫描已启动");
```

第 4 步: 验证和修复 DEX (10-30 分钟)

脱壳后的 DEX 文件可能不完整或有损坏，需要验证和修复。

验证步骤

```
# 1. 拉取导出的 DEX 文件
adb pull /data/data/com.target.app/ ./dumped_files/

# 2. 查看提取到的 DEX 文件
ls -lh ./dumped_files/*.dex
# 输出示例:
# -rw-r--r-- 1 user user 5.2M dumped_1234567890.dex
# -rw-r--r-- 1 user user 1.8M dumped_0987654321.dex

# 3. 验证 DEX 文件完整性
xxd ./dumped_files/dumped_1234567890.dex | head -n 2
# 应该看到 DEX magic: 64 65 78 0a (dex\n)

# 4. 使用 JADX 打开验证
jadx ./dumped_files/dumped_1234567890.dex
# 如果能正常反编译，表示脱壳成功
```

常见需要修复的情况

1. 方法体被 NOP 填充:

- 症状: JADX 反编译后看到大量空方法或只有 `return` 的方法
- 原因: 壳用占位符替换了真实代码
- 解决: 如果用 FART dump 了 CodeItem，需要手动替换回去

2. 字符串池损坏:

- 症状: 反编译后字符串显示为乱码或缺失
- 解决: 使用 `dex-repair` 工具重建字符串池

3. 类/方法索引错乱:

- 症状: 方法调用关系不正确
- 解决: 使用 `smali/baksmali` 重新组装

自动化修复工具

```
# 使用dex-repair (开源工具)
git clone https://github.com/F8LEFT/dex-repair
cd dex-repair
python3 repair.py ./dumped_files/dumped_1234567890.dex -o ./fixed.dex

# 验证修复结果
jadx ./fixed.dex
```

手动修复流程 (FART 方法抽取)

```
# 1. 将导出的 CodeItem 替换回 DEX
# 这需要使用 DexPatcher 或自定义脚本

# 2. 反汇编 DEX
baksmali d dumped_1234567890.dex -o ./smali_output

# 3. 查找空方法并替换
# 在 smali_output 中, 找到方法体为空的 .smali 文件
# 将导出的 CodeItem 反汇编后的内容复制进去

# 4. 重新组装
smali a ./smali_output -o ./repacked.dex

# 5. 验证
jadx ./repacked.dex
```

原理深入

加固流程示意

打包时加固流程：



运行时解密流程：

```
Application.onCreate()
    ↓
    壳代码执行
    ↓
    解密 encrypted.dat
    ↓
    DexClassLoader.load(解密后的 DEX)
    ↓
    跳转到原始 Application 入口
```

脱壳时机示意

```
App 启动
    ↓
    壳代码运行
    ↓
    解密原始 DEX ← Hook 点 1: ClassLoader
    ↓
    ART 加载 DEX 到内存 ← Hook 点 2: libart.so
    ↓
    编译为 OAT 格式
    ↓
    类初始化和方法调用 ← Hook 点 3: ArtMethod::Invoke
    ↓
    原始代码执行
```

常见问题

✖ 问题 1: frida-dexdump 无法 dump 任何文件

可能原因:

1. 壳检测到 Frida 并提前退出
2. Hook 时机太晚, DEX 已经加载完毕
3. 使用了非标准的加载方式

解决方案:

```
# 1. 先绕过 Frida 检测
frida -U -f com.target.app -l bypass_frida_detection.js --no-pause

# 等待应用启动后, 再运行 dexdump (分两步)
frida -U com.target.app -l frida_dexdump_manual.js

# 2. 尝试更早的拦截点
# 修改 frida-dexdump 源码, 在 libc.so fork() 之前就注入

# 3. 使用内存扫描作为兜底方案
frida -U com.target.app -l unpacker_memscan.js
```

✖ 问题 2: Dump 的 DEX 无法被 JADX 识别

可能原因:

1. Dump 的时机不对, DEX 还未完全解密
2. DEX 文件被截断
3. 内存中的 DEX 已被修改 (如方法抽取)

解决方案:

```
# 1. 检查 DEX 文件头  
xxd dumped.dex | head -n 5  
# 前 4 字节必须是: 64 65 78 0a (dex\n)  
# 后 4 字节是版本号: 30 33 35 00 (035) 或 30 33 38 00 (038)  
  
# 2. 验证文件大小  
# 偏移 32 字节处记录文件大小  
dd if=dumped.dex bs=1 skip=32 count=4 | xxd  
# 与实际文件大小对比  
  
# 3. 尝试修复工具  
dex-repair dumped.dex -o fixed.dex  
  
# 4. 如果是方法抽取壳, 需要用 FART 补全方法体
```

✖ 问题 3: FART 脚本导致应用崩溃

可能原因:

1. ArtMethod 结构偏移错误 (Android 版本不匹配)
2. 读取了无效的内存地址
3. Hook 符号错误

解决方案:

```
// 1. 添加异常保护
Interceptor.attach(artMethodInvokeAddr, {
    onEnter: function (args) {
        try {
            var artMethod = args[0];
            // ... 你的代码
        } catch (e) {
            console.log("[-] 捕获异常: " + e);
            // 不要重新抛出, 避免崩溃
        }
    },
});

// 2. 根据 Android 版本动态调整偏移
var androidVersion = Java.androidVersion;
var codeItemOffset;
if (androidVersion >= 10) {
    codeItemOffset = 16; // Android 10+
} else if (androidVersion >= 8) {
    codeItemOffset = 20; // Android 8-9
} else {
    codeItemOffset = 24; // Android 7
}

// 3. 检查指针有效性
if (codeItemPtr && !codeItemPtr.isNull()) {
    // 尝试读取前先检查是否可读
    try {
        Process.findRangeByAddress(codeItemPtr); // 会抛出异常如果地址无效
        var insnsSize = codeItemPtr.add(12).readU32();
        // ...
    } catch (e) {
        console.log("[-] 无效地址: " + codeItemPtr);
    }
}
```

✖ 问题 4: 方法抽取壳只 dump 出部分方法

可能原因:

1. FART 技术只能 dump 被调用过的方法
2. 部分方法在特定条件下才会触发

解决方案:

```
# 1. 使用 FART 技术（见第 3 步方法 C）
# 必须触发所有关键方法调用才能完整导出

# 2. 手动触发方法调用
# 写一个测试脚本，遍历所有类的所有方法并调用

# 3. 使用专用工具
# FUPK3、Youpk 等工具已内置方法主动调用逻辑
```

主动调用所有方法的脚本：

```
Java.perform(function () {
    Java.enumerateLoadedClasses({
        onMatch: function (className) {
            if (className.indexOf("com.target.app") !== -1) {
                try {
                    var clazz = Java.use(className);
                    var methods = clazz.class.getDeclaredMethods();

                    methods.forEach(function (method) {
                        try {
                            // 尝试调用静态方法（传空参数）
                            console.log("[+] 尝试调用: " + method.getName());
                            method.invoke(null, []);
                        } catch (e) {
                            // 忽略调用失败
                        }
                    });
                } catch (e) {}
            }
        },
        onComplete: function () {
            console.log("[+] 方法触发完成");
        },
    });
});
```

相关链接

相关 Recipe

项目	说明
Recipe: 绕过 App 对 Frida 的检测	脱壳前通常需要先过反调试
Recipe: 抓包分析 Android 应用的网络流量	脱壳后抓包分析加密逻辑
Recipe: SO 混淆与反混淆	Native 层加固的处理

工具深入

- [Frida 内部原理](#) - 理解 Frida Hook 机制
- [Unidbg 使用指南](#) - 仿真执行 Native 解密函数

案例分析

- [案例: 某音乐 App 的加固分析](#)

参考资料

- [DEX 文件格式详解](#)
- [ART 运行时机制](#)

快速参考

加固检测速查表

检测项	命令	可疑特征
Application 入口	<code>unzip -p app.apk AndroidManifest.xml \ grep android:name</code>	<code>StubShell</code> , <code>ApplicationWrapper</code> , <code>StubApplication</code>
DEX 文件大 小	<code>unzip -l app.apk \ grep classes.dex</code>	< 100 KB (复杂应用)
加密数据文件	<code>unzip -l app.apk \ grep -E "\.dat\ \.bin"</code>	<code>assets/</code> 下的 .dat/.bin 文件
可疑 SO 库	<code>unzip -l app.apk \ grep "lib/*\.so"</code>	<code>libexec.so</code> , <code>libvmp.so</code> , <code>libprotect.so</code>
使用 PKid	<code>java -jar ApkToolPlus.jar -pkid app.apk</code>	直接输出加固厂商

常用脱壳命令

```
# 1. 使用 frida-dexdump (推荐)
python3 -m frida_dexdump -U -f com.target.app -o ./output

# 2. 手写脚本 (Spawn 模式)
frida -U -f com.target.app -l unpacker.js --no-pause

# 3. 内存扫描
frida -U com.target.app -l memscan.js

# 4. 拉取导出文件
adb pull /data/data/com.target.app/ ./dumped/

# 5. 验证 DEX 文件
xxd dumped.dex | head -n 2 # 检查 magic number
jad dumped.dex # 尝试反编译
```

常用工具

工具	用途	链接
frida-dexdump	自动化 DEX dumper	GitHub
FUPK3	针对特定壳的脱壳机	GitHub
Youpk	较新的脱壳工具	GitHub
PKid	加固识别工具	GitHub
dex-repair	DEX 文件修复工具	GitHub

 提示：脱壳是一个需要耐心和经验的过程。如果一种方法不奏效，尝试组合多种技术。记住，代码运行必解密 - 只要应用能正常运行，理论上就能脱壳。

[R13] Frida 脱壳与 SO 修复

R13: 使用 Frida 脱壳加固 App 并修复 SO 文件

问题场景

你遇到了什么问题？

- 用 jadx 打开 APK，发现代码被混淆或看不到关键逻辑
- 🛡️ APK 使用了加固（加壳）保护，无法静态分析
- 📦 SO 文件被加密，IDA Pro 无法正确加载
- 你想获取 App 运行时真正的 DEX 文件
- 🌿 需要分析 Native 层代码，但 SO 文件已被加壳

本配方教你：使用 Frida 动态脱壳加固 App、Dump 内存中的 DEX 和 SO 文件、修复文件格式以供分析。

核心理念：

💡 动态脱壳：在运行时获取已解密的代码

- 加壳只是静态保护，运行时必然会解密
- Frida 可以在 DEX/SO 加载时 dump 内存
- 修复文件格式后即可用传统工具分析
- 绕过所有加固方案的通用方法

预计用时：30-60 分钟

工具清单

必需工具

- Frida - 动态插桩框架
- Android 设备（已 Root）或模拟器
- Python 3.7+ - 运行 Frida 脚本
- jadx-gui - 分析脱壳后的 DEX

可选工具

- IDA Pro / Ghidra - 分析 SO 文件
- frida-dexdump - 自动化 DEX 脱壳
- FRIDA-DEXDump - 另一个流行的脱壳工具
- SoFixer - 修复 dump 的 SO 文件

前置条件

✓ 确认清单

```
# 1. Frida 正常运行  
frida-ps -U  
  
# 2. Python 环境  
python3 --version  
  
# 3. jadx-gui 已安装  
jadx-gui --version  
  
# 4. 检查设备 root 状态  
adb shell su -c 'id'  
# 应该显示 uid=0(root)
```

识别 App 是否加固

方法 1: jadx 查看

打开 APK, 如果看到:

- 只有几个类和方法
- 有 `StubApp`、`ProxyApplication` 等字样
- `MainActivity` 逻辑异常简单

方法 2: 查看 SO 文件

```
# 解压 APK  
unzip app.apk -d app_unzipped  
  
# 查看 lib 目录  
ls app_unzipped/lib/arm64-v8a/  
  
# 常见加固壳 SO 文件名  
# libjiagu.so (360加固)  
# libDexHelper.so (梆梆加固)  
# libtup.so (腾讯加固)  
# libexec.so (爱加密)
```

解决方案

第 1 步: 使用 frida-dexdump 脱壳 (10 分钟)

1.1 安装 frida-dexdump

```
# 克隆项目  
git clone https://github.com/hluwa/frida-dexdump.git  
cd frida-dexdump  
  
# 安装依赖  
pip3 install frida frida-tools
```

1.2 运行脱壳

```
# -f: 启动应用  
# --no-pause: 不暂停, 立即运行  
python3 main.py -U -f com.example.app  
  
# 脚本会自动:  
# 1. 启动应用  
# 2. Hook DEX 加载函数  
# 3. 导出所有已加载的 DEX 文件  
# 4. 保存到当前目录
```

输出示例:

```
[DEXDump] Dumping DEX file: 0x7abc000000, size: 4562314  
[DEXDump] Saved: com.example.app_classes.dex  
[DEXDump] Found DEX: /data/app/.../base.apk!classes2.dex  
[DEXDump] Dumping DEX file: 0x7abc500000, size: 2314567  
[DEXDump] Saved: com.example.app_classes2.dex  
[DEXDump] Total: 2 DEX files dumped
```

1.3 验证脱壳结果

```
# 用 jadx 打开  
jadx-gui com.example.app_classes.dex
```

第 2 步：使用 FRIDA-DEXDump（备选方案）（10 分钟）

如果 frida-dexdump 不工作，可以尝试 FRIDA-DEXDump：

```
# 安装  
git clone https://github.com/lasting-yang/frida_dump.git  
cd frida_dump  
  
# 运行命令  
python3 dump_dex.py -U -f com.example.app
```

第 3 步：手动脚本脱壳（高级）（15 分钟）

如果自动化工具失败，可以编写自定义 Frida 脚本。

3.1 Hook OpenCommon（适用于 Android 8.0+）

dex_dump.js:

```
function dumpDex() {
    Java.perform(function () {
        console.log("[*] DEX Dumper started");

        // 查找 libart.so
        var libart = Process.findModuleByName("libart.so");
        if (!libart) {
            console.log("[-] libart.so not found");
            return;
        }

        // Hook OpenCommon (Android 8.0+)
        // 符号名称因版本而异，需要用 nm 或 readelf 确认
        var OpenCommon = null;

        // 尝试常见符号
        var symbols = [
            "_ZN3art7DexFile10OpenCommonEPKhjS2_jRKNS_100atDexFileEbbPS1_",
            "_ZN3art7DexFile10OpenCommonEPKhmS2_jRKNS_100atDexFileEbbPS1_NS_6Handle",
        ];

        for (var i = 0; i < symbols.length; i++) {
            OpenCommon = Module.findExportByName("libart.so", symbols[i]);
            if (OpenCommon) {
                console.log("[+] Found OpenCommon:", OpenCommon);
                break;
            }
        }

        if (!OpenCommon) {
            console.log("[-] OpenCommon not found");
            return;
        }

        // Hook
        Interceptor.attach(OpenCommon, {
            onEnter: function (args) {
                // args[0] = base (DEX 内存地址)
                // args[1] = size (DEX 文件大小)

                var base = args[0];
                var size = args[1].toInt32();

                console.log("[*] 检测到 DEX!");
                console.log("    基址: " + base);
                console.log("    大小: " + size);

                // 读取 DEX 文件头，验证魔数
                var magic = base.readCString(4);
                if (magic === "dex\n") {
                    console.log("    Magic: " + magic + " ✓");
                }
            }
        });
    });
}
```

```
// Dump DEX
var dexBytes = base.readByteArray(size);
var fileName = "/sdcard/" + size + ".dex";

var file = new File(fileName, "wb");
file.write(dexBytes);
file.close();

console.log("[+] DEX dumped to: " + fileName);
} else {
    console.log("    Invalid magic: " + magic);
}
},
});

console.log("[*] Hooks installed, waiting for DEX load...");  

});  

}

setImmediate(dumpDex);
```

使用方法：

```
# 运行脚本
frida -U -f com.example.app -l dex_dump.js --no-pause

# 拉取到本地
adb pull /sdcard/*.dex .
```

3.2 查找正确的符号名

```
Module.enumerateExports("libart.so").forEach(function (exp) {
    if (exp.name.includes("DexFile") && exp.name.includes("Open")) {
        console.log(exp.name, exp.address);
    }
});
```

第 4 步: Dump SO 文件 (10 分钟)

4.1 查看已加载的 SO

```
# 查看进程加载的 SO 文件  
frida -U -f com.example.app
```

在 REPL 中输入：

```
Process.enumerateModules().forEach(function (m) {  
    if (m.name.includes("native") || m.name.includes("encrypt")) {  
        console.log(m.name, m.base, m.size);  
    }  
});  
  
// 输出示例：  
// libnative-lib.so 0x7abc000000 0x50000
```

4.2 Dump SO 内存

```
function dumpSo(moduleName) {  
    var module = Process.findModuleByName(moduleName);  
    if (!module) {  
        console.log("[-] Module not found: " + moduleName);  
        return;  
    }  
  
    console.log("[+] 找到模块:", moduleName);  
    console.log("    基址: " + module.base);  
    console.log("    大小: " + module.size);  
  
    // 导出整个模块  
    var buffer = module.base.readByteArray(module.size);  
    var fileName = "/sdcard/" + moduleName;  
  
    var file = new File(fileName, "wb");  
    file.write(buffer);  
    file.close();  
  
    console.log("[+] 已导出到: " + fileName);  
}  
  
// 使用  
dumpSo("libnative-lib.so");
```

拉取文件：

```
adb pull /sdcard/libnative-lib.so .
```

4.3 使用 frida-all-in-one (推荐)

```
# 克隆项目  
git clone https://github.com/hookmaster/frida-all-in-one.git  
cd frida-all-in-one  
  
# 运行命令  
python3 dump_so.py -U com.example.app libnative-lib.so  
  
# 会自动导出并修复 SO 文件
```

第 5 步：修复 SO 文件（10 分钟）

从内存 dump 的 SO 文件可能缺少 ELF 头信息，需要修复。

5.1 使用 SoFixer

```
# 下载  
git clone https://github.com/F8LEFT/SoFixer.git  
cd SoFixer  
  
# 编译 (需要 CMake)  
mkdir build && cd build  
cmake ..  
make  
  
# 使用  
../SoFixer ../libnative-lib.so ../libnative-lib_fixed.so
```

输出示例：

```
[+] Detected architecture: ARM64
[+] Rebuilding ELF header...
[+] Fixing section table...
[+] Fixing dynamic symbols...
[+] Output file: libnative-lib_fixed.so
[+] Done!
```

5.2 验证修复结果

```
# 检查文件类型
file libnative-lib_fixed.so
# 应该显示: ELF 64-bit LSB shared object, ARM aarch64...

# 用 IDA Pro 打开
# 或用 readelf 查看
readelf -h libnative-lib_fixed.so
```

原理深入

DEX 脱壳原理



OpenCommon 函数签名

```
// Android 8.0+ 的 OpenCommon 签名（简化）
static std::unique_ptr<DexFile> OpenCommon(
    const uint8_t* base, // DEX 内存基址
    size_t size, // DEX 大小
    ...
)
```

SO Dump 原理

内存中的 SO 布局：

- [ELF Header] ← 可能被壳破坏
- [.text 段]
- [.data 段]
- [.rodata 段]
- ...
- [Symbol Table] ← 需要重建
- [String Table] ← 需要重建

直接 dump 内存只能获取段数据，缺少完整 ELF 结构，所以需要 SoFixer 修复。

常见加固壳识别

加固厂商	SO 文件名	特点
360 加固	libjiagu.so	整体加密
梆梆加固	libDexHelper.so	方法抽取
腾讯加固	libtup.so	VMP 保护
爱加密	libexec.so	多层次加密
网易易盾	libnesec.so	云端保护

通用策略：所有加固都需要在运行时解密，Frida 脱壳对所有方案都有效！

常见问题

✖ 问题 1: frida-dexdump 报错 "Failed to spawn"

症状：无法启动应用

解决：

```
# 1. 确认应用已安装  
adb shell pm list packages | grep example  
  
# 2. 确认包名正确  
# 从 AndroidManifest.xml 获取准确包名  
  
# 3. 尝试 Attach 模式  
# 先手动启动应用  
adb shell am start -n com.example.app/.MainActivity  
  
# 再附加  
python3 main.py -U com.example.app
```

✖ 问题 2: Dump 的 DEX 无法用 jadx 打开

可能原因：DEX 头部损坏

解决步骤：

1. 检查魔数

```
xxd dumped.dex | head -1  
# 应该看到: 64 65 78 0a (dex\n)
```

1. 验证 DEX 大小

```
# 验证 DEX 大小
with open('dumped.dex', 'rb') as f:
    f.seek(32) # 跳到 file_size 字段
    size = int.from_bytes(f.read(4), 'little')
    print(f"DEX 声明的大小: {size}")

import os
actual_size = os.path.getsize('dumped.dex')
print(f"实际文件大小: {actual_size}")
```

1. 使用 dexrepair 修复

```
git clone https://github.com/anestisb/dexrepair.git
python3 dexrepair/dexrepair.py dumped.dex fixed.dex
```

✗ 问题 3: Hook 点没有触发

检查步骤:

1. 确认 libart.so 已加载

```
var libart = Process.findModuleByName("libart.so");
console.log("libart found:", libart !== null);
```

1. 列出所有 OpenCommon 符号

```
Module.enumerateExports("libart.so").forEach(function (exp) {
    if (exp.name.includes("OpenCommon")) {
        console.log(exp.name);
    }
});
```

1. 尝试其他 Hook 点

```
// Android 7.0-
var OpenMemory = Module.findExportByName(
    "libart.so",

    "_ZN3art7DexFile100openMemoryEPKhjRKNSt3__112basic_stringIcNS3_11char_traitsIcEENS3_9al
locatorIcEEEEjPNS_6MemMapEPKNS_100atDexFileEPS9_"
);
```

✖ 问题 4: SO 文件修复后 IDA 仍无法分析

症状: IDA 打开后只显示数据, 没有函数

解决:

1. 手动定义函数

在 IDA 中:

- 光标移到疑似函数起始处
- 按 'P' 键创建函数
- 按 'C' 键转换为代码

1. 使用符号恢复工具

```
# 如果原始 SO 有符号表
readelf -s original.so > symbols.txt

# 用 IDA 脚本导入符号
```

1. 检查是否有 OLLVM 混淆

- 如果看到大量跳转和无意义的代码块
- 可能是 OLLVM 控制流平坦化
- 参考: [OLLVM 反混淆](#)

✖ 问题 5: App 检测到 Frida 并崩溃

症状: 启动后立即退出, logcat 显示反调试提示

解决：

参考 [Frida 反调试绕过](#)

快速方法：

```
# 使用 Magisk Hide  
# 或使用修改版 Frida 服务器  
wget https://github.com/hluwa/strongR-frida-android/releases/download/xxx/frida-server
```

相关链接

相关 Recipe

- [应用脱壳总览](#) - 各种脱壳技术对比
- [Frida 反调试绕过](#) - 处理反 Frida 检测
- [SO 混淆分析](#) - 分析混淆的 SO 文件
- [OLLVM 反混淆](#) - 处理控制流混淆

工具深入

- [Frida 完整指南](#)
- [IDA Pro 使用](#)

项目资源

项目	说明	链接
frida-dexdump	自动化 DEX 脱壳	https://github.com/hluwa/frida-dexdump
FRIDA-DEXDump	深度 DEX 脱壳	https://github.com/lasting-yang/frida_dump
SoFixer	SO 文件修复	https://github.com/F8LEFT/SoFixer
frida-all-in-one	综合工具集	https://github.com/hookmaster/frida-all-in-one

理论基础

- DEX 文件格式
- SO/ELF 文件格式
- ART 运行时

快速参考

脱壳工具对比

工具	类型	难度	特点
frida-dexdump	自动化	简单	简单, 支持多版本
FRIDA-DEXDump	自动化	简单	深度搜索, 更全面
手动脚本	定制	中等	灵活, 适合特殊情况
objection	交互式	简单	多功能, 含脱壳

一键脱壳脚本

auto_unpack.sh:

```
#!/bin/bash

PACKAGE=$1

if [ -z "$PACKAGE" ]; then
    echo "用法: $0 <package_name>"
    exit 1
fi

echo "🔒 开始脱壳: $PACKAGE"

# 1. Dump DEX
echo ""
echo "📦 导出 DEX 文件..."
python3 ~/tools/frida-dexdump/main.py -U -f $PACKAGE

# 2. Dump SO
echo ""
echo "📚 导出 SO 文件..."
frida -U -f $PACKAGE -l dump_all_so.js --no-pause

sleep 5

# 3. 拉取文件
echo ""
echo "👉 拉取文件..."
adb pull /sdcard/*.dex .
adb pull /sdcard/*.so .

# 4. 清理
adb shell rm /sdcard/*.dex
adb shell rm /sdcard/*.so

echo ""
echo "✅ 完成! 文件已保存到当前目录"
ls -lh *.dex *.so
```

dump_all_so.js

```
function dumpAllSo() {
    var modules = Process.enumerateModules();
    console.log("[*] 找到 " + modules.length + " 个模块");

    modules.forEach(function (module) {
        // 只导出 .so 文件
        if (!module.name.endsWith(".so")) {
            return;
        }

        // 排除系统库
        if (module.path.startsWith("/system") || module.path.startsWith("/apex")) {
            return;
        }

        console.log("[+] 导出: " + module.name);
        console.log("    路径: " + module.path);
        console.log("    基址: " + module.base);
        console.log("    大小: " + module.size);

        try {
            var buffer = module.base.readByteArray(module.size);
            var fileName = "/sdcard/" + module.name;
            var file = new File(fileName, "wb");
            file.write(buffer);
            file.close();
            console.log("    已保存: " + fileName);
        } catch (e) {
            console.log("    错误: " + e);
        }
    });

    console.log("[*] 完成!");
}

setImmediate(dumpAllSo);
```

 提示：脱壳是一个反复试错的过程。如果一种方法不起作用，尝试其他方法或工具。大多数加固方案都可以通过 Frida 动态脱壳绕过。

[R14] SO 混淆与反混淆

R14: SO 文件反混淆：花指令识别与自动化去除

在 Android SO 文件逆向工程中，代码混淆 (Code Obfuscation)，俗称“花指令”，是开发者为了保护核心逻辑、增加逆向分析难度而采用的一种常用技术。其核心思想是在代码中插入大量对程序本身逻辑无用但能迷惑反汇编工具和分析人员的指令。

本指南将系统介绍花指令的常见类型、识别方法，并重点阐述如何利用 `IDAPython` 编写脚本，实现自动化“去花”。

目录

- 花指令的核心类型
 - 垃圾指令 (Junk Code)
 - 不透明谓词 (Opaque Predicates)
 - 控制流平坦化 (Control Flow Flattening)
- 如何识别花指令
- 自动化去花脚本 (`IDAPython` 实战)

花指令的核心类型

垃圾指令 (Junk Code)

最简单的混淆形式。在真实指令之间插入不会影响程序状态（寄存器、内存、标志位）的指令。

```
; 真实代码  
PUSH EAX  
  
; --- 垃圾代码 ---  
NOP  
MOV EBX, EBX  
XCHG ECX, ECX  
ADD EAX, 0  
; --- 垃圾代码结束 ---  
  
; 真实代码  
POP EAX
```

不透明谓词 (Opaque Predicates)

通过构造条件恒成立或恒不成立的分支，让反汇编器和分析人员误以为存在多个执行路径。

```
MOV EAX, EDX  
XOR EAX, EDX      ; 清零 EAX  
TEST EAX, EAX     ; 设置 Z 标志  
  
; JZ (为零则跳转) 将始终跳转  
; JNZ 分支下的代码是永远不会执行的死代码  
JZ real_code_path  
; --- 死代码 ---  
ADD EAX, 1234  
CALL some_fake_func  
; --- 死代码结束 ---  
  
real_code_path:  
; ... 真实代码
```

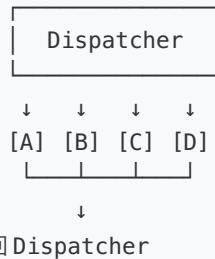
控制流平坦化 (Control Flow Flattening)

这是一种高级且非常有效的混淆技术。它将一个函数的正常逻辑块打散，然后使用一个中央分发器 (Dispatcher) 和 `switch-case` 结构来控制执行流。原始的调用关系被隐藏在一个巨大的循环中，使得函数逻辑极难被理解。

原始代码：

A → B → C → D

平坦化后：



如何识别花指令

静态分析特征

在 IDA Pro 或 Ghidra 中观察：

- 无效跳转：`JMP loc_A` 的下一条指令就是 `loc_A`
- 跳转到指令中间：`JMP $+5` 跳转到一个正常指令的中间，破坏反汇编
- 对称操作：连续的 `PUSH / POP` 同一个寄存器
- 恒成立/不成立的条件：在 `Jcc` 指令前，`CMP` 的两个操作数明显相等或不等
- 无意义的计算：计算结果没有被后续代码使用
- IDA 图形视图：控制流平坦化的函数会呈现出一个巨大的、节点众多的 `switch` 结构，所有逻辑块都指向一个中心分发块

动态调试验证

最可靠的方法。使用 `gdb` 或 IDA 的调试器：

- 在可疑分支下断点，如果断点从未命中，则说明该分支是死代码
- 单步执行，观察寄存器和内存的变化。如果一段指令执行后，相关状态没有变化，则很可能 是垃圾代码

自动化去花脚本 (IDAPython 实战)

当花指令数量庞大时，手动修复是不现实的。编写脚本自动化处理是唯一高效的途径。以下以 IDAPython 为例。

场景一：NOP 掉无效跳转

一个常见的花指令模式是 `JMP dest`，而 `dest` 紧接着 `JMP` 指令。

```
.text:00001234 JMP short loc_1236 ; 跳转指令本身占 2 字节  
.text:00001236 ; ... 真实代码
```

IDAPython 脚本：

```
import idaapi
import idc
import idautils

def patch_junk_jumps():
    """
    查找并将形式为 `JMP next_instruction` 的跳转 NOP 掉。
    """

    print("扫描垃圾跳转...")
    count = 0

    # 遍历代码段
    for seg_ea in idautils.Segments():
        if idc.get_segm_attr(seg_ea, idc.SEGATTR_TYPE) != idc.SEG_CODE:
            continue

        seg_start = idc.get_segm_start(seg_ea)
        seg_end = idc.get_segm_end(seg_ea)

        for head in idautils.Heads(seg_start, seg_end):
            # 检查是否是 JMP 指令
            if idaapi.is_jmp_insn(head):
                # 获取跳转目标地址
                target_ea = idc.get_operand_value(head, 0)
                # 获取指令长度
                insn_len = idc.get_item_size(head)

                # 如果跳转目标是下一条指令的地址
                if target_ea == (head + insn_len):
                    print(f"在 0x{head:X} 处找到垃圾 JMP, 目标: 0x{target_ea:X}")

                # 用 NOP 修补
                for i in range(insn_len):
                    idc.patch_byte(head + i, 0x90)
                count += 1

    print(f"完成。修补了 {count} 个垃圾跳转。")

# 执行脚本
patch_junk_jumps()
```

场景二：识别并去除不透明谓词

```
import idc
import idautils

def patch_opaque_predicates():
    """
    查找并去除不透明谓词。
    """

    print("扫描不透明谓词...")
    count = 0

    for seg_ea in idautils.Segments():
        if idc.get_segm_attr(seg_ea, idc.SEGATTR_TYPE) != idc.SEG_CODE:
            continue

        seg_start = idc.get_segm_start(seg_ea)
        seg_end = idc.get_segm_end(seg_ea)

        for head in idautils.Heads(seg_start, seg_end):
            # 检查是否是 JNE 指令（例如，操作码 0x75）
            if idc.get_byte(head) == 0x75:
                # 检查 JNE 之前的指令是否是 CMP
                prev_head = idc.prev_head(head)
                if idc.print_insn_mnem(prev_head) == "cmp":
                    # 检查 CMP 的两个操作数是否相同
                    op1 = idc.get_operand_value(prev_head, 0)
                    op2 = idc.get_operand_value(prev_head, 1)

                    # 这是一个简化示例，实际的操作数类型检查会更复杂
                    if op1 == op2: # 例如, CMP EAX, EAX
                        print(f"在 0x{head:X} 处找到不透明谓词")
                        # 将 JNE 指令 NOP 掉
                        insn_len = idc.get_item_size(head)
                        for i in range(insn_len):
                            idc.patch_byte(head + i, 0x90)
                        count += 1

    print(f"完成。修补了 {count} 个不透明谓词。")

# 执行脚本
patch_opaque_predicates()
```

通用去花流程

1. 观察：在 IDA Pro 中观察可疑代码的模式
2. 识别：找到该模式的通用机器码或指令特征
3. 编码：编写脚本，精确地定位这些特征并进行修复 (Patch)

虽然花指令的变种层出不穷，但其本质是有限的。掌握了自动化的脚本去花能力，就能极大地提升 SO 文件逆向分析的效率。

相关链接

相关 Recipe

- [OLLVM 反混淆 - 处理控制流平坦化](#)
- [应用脱壳总览 - 脱壳后再去花](#)

工具深入

- [IDA Pro 使用指南](#)
- [Ghidra 使用指南](#)

 提示：去花是一个需要耐心和经验的过程。建议先手动分析几个样本，理解混淆模式后再编写脚本自动化处理。

[R15] SO 字符串解密

R15: SO 文件字符串混淆对抗指南

在 Android Native 层安全中，字符串混淆是一种用于隐藏敏感信息、增加逆向分析难度的常用技术。开发者通过对 SO 文件中的关键字符串（如 API URL、加密密钥、Shell 命令、功能开关等）进行编码或加密，可以有效防止静态分析工具（如 `strings` 命令或 IDA Pro 的字符串窗口）直接发现它们。

本文旨在系统性地介绍 SO 文件中常见的字符串混淆技术，并提供一套从静态分析到动态分析的完整对抗策略。

目录

- SO 文件字符串混淆对抗指南
 - 目录
 - 字符串混淆的核心思想
 - 常见的混淆技术
 - 对抗策略一：静态分析 (IDA Pro / Ghidra)
 - 识别解密/解混淆函数
 - 定位交叉引用
 - 自动化脚本解密
 - 示例: Python 实现的简单 XOR 解密算法 - 内存漫游与搜索
 - 总结：最高效的分析流程

字符串混淆的核心思想

其本质是避免将明文字符串直接存储在二进制文件的 `.rodata` 或 `.data` 段中。取而代之的是，在程序运行时，通过特定的函数动态地在内存中（栈或堆）恢复出原始的字符串。

一个典型的流程如下：`加密的字节数组` -> `解密/解混淆函数` -> `内存中的明文字符串`

我们的目标就是截获“内存中的明文字符串”。

常见的混淆技术

1. 简单编码：

- Base64: 将 Base64 编码后的字符串存储，使用时再解码。
- ROT13/Caesar Cipher: 简单的字符位移。

1. 按位运算：

- XOR (异或): 将原始字符串与一个固定的（或动态计算的）密钥进行按字节异或。这是最常见、最高效的一种方式。

1. 栈上构建：

- 不在任何段中存储字符串，而是在函数开始时，通过一系列 `mov` 指令逐字节地将字符串 `push` 到栈上。

```
void get_secret_string() {
    char secret[12];
    secret[0] = 's';
    secret[1] = 'e';
    // ...
    secret[10] = 't';
    secret[11] = '\0';
    // use secret
}
```

- * 使用如 AES, RC4, DES 等标准对称加密算法。密钥本身可能被再次混淆或从其他地方动态获取。

--

对抗策略一：静态分析（IDA Pro / Ghidra）

静态分析的目标是**理解解密逻辑并自动化地应用它**。

识别解密/解混淆函数

* **特征**：解密函数通常具有以下一个或多个特征：

* 接受一个指向字节数组的指针和一个长度作为参数。

* 函数内部包含一个循环结构（`for` / `while`）。

* 循环内部有按位操作，特别是 `XOR`（异或）指令。

* 函数的交叉引用（Xrefs）非常多，且调用的地方都伴随着一个数据块的地址。

* **方法**：在 IDA Pro 或 Ghidra 中，通过搜索这些代码模式，通常能很快定位到核心的解密函数。

定位交叉引用

一旦你识别出了解密函数（例如 `decrypt_string`），立即查看它的所有交叉引用。每一个调用 `decrypt_string` 的地方，都是一个加密字符串被使用的地方。传递给该函数的参数，就是加密的数据。

自动化脚本解密

这是静态分析的精髓所在。

1. **分析算法**：仔细阅读解密函数的汇编或反编译代码，用一种高级语言（如 Python）重新实现其逻辑。

```
```python
示例：Python 实现的简单 XOR 解密算法
def decrypt_xor(data, key):
 decrypted = bytearray()
 for i in range(len(data)):
 decrypted.append(data[i] ^ key[i % len(key)])
 return decrypted.decode('utf-8')
```

- 脚本逻辑：

1. 获取解密函数的地址。
2. 遍历该函数的所有交叉引用。
3. 在每个交叉引用的地方，解析其参数，提取出加密数据块的地址和长度。
4. 读取加密数据。
5. 调用步骤 1 中实现的 Python 解密函数。

6. 将解密后的明文字符串，作为注释，添加到交叉引用的代码行旁边。

- 效果：运行脚本后，IDA/Ghidra 中的代码将变得非常易读，所有加密字符串都以注释的形式被“还原”了。

## 对抗策略二：动态分析 (Frida)

动态分析的核心思想是不关心解密过程，只关心解密结果。它通常更快速、更直接。

### Hook 解密函数（首选策略）

这是对抗字符串混淆最简单、最高效的方法。

1. 定位函数：使用静态分析工具（IDA/Ghidra）找到解密函数的地址。

2. 编写 Frida 脚本：

- Hook `onEnter`：在进入解密函数时，打印其输入参数（加密的字节数组）。
- Hook `onLeave`（更常用）：在函数返回时，直接读取其返回值。因为返回值通常就是指向内存中明文字符串的指针。

```
const decryptFuncPtr = Module.findExportByName(
 "libnative-lib.so",
 "Java_com_example_MainActivity_decryptString"
);
// 或者直接使用地址: const decryptFuncPtr = Module.getBaseAddress("libnative-
lib.so").add(0x1234);
```

```
Interceptor.attach(decryptFuncPtr, { onEnter: function(args) { console.log("进入
decryptString, 数据: " + args[0].readCString()); }, onLeave: function(retval) { // retval
是指向解密后字符串的指针 var decryptedString = retval.readCString(); console.log("解密
后的字符串 -> " + decryptedString); // 可以进一步将结果写入文件 // send({ decrypted:
decryptedString }); } });
```

#### ##### 内存漫游与搜索

在某些情况下，App 可能会在启动时一次性解密大量字符串，并将它们存放在一个特定的内存区域。

\* \*\*方法\*\*：

1. 让 App 运行一段时间。
2. 使用 Frida 的 `Memory.scan` API 在进程的整个内存空间中搜索你感兴趣的字符串模式（例如，`https://`）。

```
```javascript
Memory.scan(Process.findModuleByName("libnative-lib.so").base,
Process.findModuleByName("libnative-lib.so").size, "68 74 74 70 70 73 3a 2f 2f", { // 十六
进制表示 "https://"
onMatch: function(address, size) {
console.log("在以下地址找到模式: " + address);
// 可能需要回退一些字节来找到字符串的起始位置
console.log(address.readCString());
},
onComplete: function() {
console.log("内存扫描完成。");
}
});
```

```

## 总结：最高效的分析流程

对于字符串混淆，最高效的工作流程是结合静态和动态分析：

1. 静态分析定位：使用 IDA Pro 或 Ghidra 快速浏览 SO 文件，识别出可能的解密/解混淆函数。
2. 动态分析验证/获取：使用 Frida 对上一步定位到的函数地址进行 Hook，运行 App 并观察 `onLeave` 的返回值，快速获取所有解密后的字符串。
3. (可选) 静态分析脚本化：如果需要对大量字符串进行离线分析或希望得到一个带注释的、更易读的反汇编文件，再回到静态分析工具中，根据已知的算法编写自动化解密脚本。

## 分析技术

---

## [R16] 逆向工程流程

# R16: Android 应用逆向工程完整工作流程

## 问题场景

你刚拿到一个 Android 应用需要分析，但面临以下挑战：

- "拿到 APK 后应该先做什么？从哪里入手？"
- "静态分析和动态分析应该如何配合？"
- "如何系统化地分析，而不是盲目尝试？"
- "遇到加固、混淆、反调试该怎么办？"
- "分析完成后如何修改应用以达到目的？"

本配方提供一个经过实战验证的标准化工作流程，帮助你系统化地完成从信息收集到代码修改的整个逆向工程过程。

## 工具清单

### 必备工具

| 项目          | 说明                    |
|-------------|-----------------------|
| APK 提取      | ADB + Package Manager |
| 解包/回包       | Apktool               |
| 反编译工具       | Jadx-GUI（推荐）或 JEB     |
| 动态分析        | Frida + Frida-tools   |
| Root 设备/模拟器 | Genymotion、夜神、雷电等     |

### 可选工具

| 项目        | 说明                               |
|-----------|----------------------------------|
| Native 分析 | IDA Pro / Ghidra / Binary Ninja  |
| 网络抓包      | mitmproxy / Burp Suite / Charles |
| 调试器       | Android Studio / jdb             |
| 签名工具      | apksigner (Android SDK 自带)       |
| 加壳检测      | PKid / ApkTool-Plus              |

## 前置知识

- 了解 Android 基本架构（四大组件、Manifest 文件）
- 掌握基本 Java/Smali 语法
- 熟悉 ADB 命令
- 拥有 Root 设备（动态分析必需）

## 解决方案

### 核心原则

由外到内、由浅入深、静动结合

1. 信息侦察 → 了解应用基本信息和技术栈
2. 静态分析 → 理解代码逻辑和算法
3. 动态验证 → 观察实际行为、绕过保护
4. 代码修改 → 实现永久性改动

### 阶段一：信息收集与初步分析（15-30 分钟）

目标：在不运行应用的情况下，快速了解基本信息、功能和潜在入口点。

## 步骤 1：获取 APK 文件

### 方法 A：从已安装应用提取

```
1. 列出所有包名
adb shell pm list packages | grep <关键词>

示例: 查找音乐应用
adb shell pm list packages | grep music
输出: package:com.example.musicapp

2. 获取 APK 路径
adb shell pm path com.example.musicapp
输出: package:/data/app/~~ABC123/com.example.musicapp-XYZ456/base.apk

3. 拉取到本地
adb pull /data/app/~~ABC123/com.example.musicapp-XYZ456/base.apk ./target.apk
```

### 一键脚本（保存为 pull-apk.sh）

```
#!/bin/bash
PACKAGE=$1
APK_PATH=$(adb shell pm path $PACKAGE | cut -d: -f2 | tr -d '\r')
adb pull $APK_PATH ./${PACKAGE}.apk
echo "[+] APK 已保存: ${PACKAGE}.apk"
```

## 步骤 2: 解包 APK

```
使用 Apktool (推荐 – 解码资源和 Smali)
apktool d target.apk -o target_unpacked

输出目录结构:
target_unpacked/
└── AndroidManifest.xml (已解码)
└── apktool.yml
└── smali/ (Dalvik 字节码)
└── smali_classes2/ (多个 DEX)
└── res/ (资源文件)
└── lib/ (native 库)
└── assets/ (资产文件)
└── original/

快速查看 (不解码)
unzip -l target.apk
unzip target.apk -d target_quick
```

## 步骤 3: 分析 AndroidManifest.xml

```
查看已解码的 manifest
cat target_unpacked/AndroidManifest.xml

或使用工具美化
xmllint --format target_unpacked/AndroidManifest.xml
```

关键信息提取表:

| 信息项             | 位置                                                             | 意义                         |
|-----------------|----------------------------------------------------------------|----------------------------|
| 包名              | <code>&lt;manifest package="..."&gt;</code>                    | 应用唯一标识                     |
| 入口 Activity     | <code>&lt;activity&gt;</code> 带 <code>LAUNCHER</code> intent   | 应用启动入口                     |
| Application 类   | <code>&lt;application android:name="..."&gt;</code>            | 自定义 Application (可能有初始化逻辑) |
| 权限              | <code>&lt;uses-permission&gt;</code>                           | 推断功能 (网络、存储、位置等)           |
| 调试标志            | <code>android:debuggable="true"</code>                         | 可直接调试                      |
| 备份标志            | <code>android:allowBackup="true"</code>                        | 数据可导出                      |
| 导出组件            | <code>android:exported="true"</code>                           | 可被外部调用                     |
| URL Scheme      | <code>&lt;intent-filter&gt;</code> 带 <code>&lt;data&gt;</code> | Deep link 入口点              |
| ContentProvider | <code>&lt;provider&gt;</code>                                  | 数据库接口                      |
| Service         | <code>&lt;service&gt;</code>                                   | 后台服务                       |

## 真实案例：分析腾讯乐固应用

```
<application
 android:name="com.tencent.StubShell.TxAppEntry" <!-- 加壳特征 -->
 android:debuggable="false"
 android:allowBackup="false">

 <activity android:name=".MainActivity"
 android:exported="true"> <!-- 可外部启动 -->
 <intent-filter>
 <action android:name="android.intent.action.MAIN"/>
 <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>

 <!-- 自定义 URL Scheme -->
 <intent-filter>
 <data android:scheme="myapp" android:host="open"/>
 <action android:name="android.intent.action.VIEW"/>
 <category android:name="android.intent.category.BROWSABLE"/>
 </intent-filter>
 </activity>
</application>
```

### 分析结论：

- 可通过 `myapp://open` URL 启动
- 调试和备份已禁用（安全配置良好）

## 步骤 4：快速目录结构审查

```
查看native 库
ls -lh target_unpacked/lib/*/
输出示例:
lib/arm64-v8a/libnative-lib.so (2.3 MB) ← Native 代码
lib/arm64-v8a/libencrypt.so (450 KB) ← 可能是加密库
lib/ararmeabi-v7a/libnative-lib.so

查看资产文件
ls -lh target_unpacked/assets/
输出示例:
config.json ← 配置文件
encrypted.dat ← 加密数据
web/index.html ← H5 页面

统计 Smali 文件数量 (估算代码规模)
find target_unpacked/smali* -name "*.smali" | wc -l
输出: 8432 (约 8000+ 类)

搜索可疑关键词
grep -r "password" target_unpacked/smali/ | head -n 10
grep -r "encrypt" target_unpacked/smali/ | head -n 10
```

阶段一产出：

- 技术栈识别（是否加壳、是否使用 native 代码）
- 潜在攻击面（导出组件、URL Scheme）
- 初步分析方向（应该深入哪里）

## 阶段二：静态分析（1-3 小时）

目标：通过反编译理解应用如何工作、算法和业务逻辑。

## 步骤 1：使用 JADX 反编译

```
启动 JADX GUI
jadx-gui target.apk

或命令行模式
jadx -d target_decompiled target.apk
```

关键搜索词：

- "encrypt"、"decrypt"、"AES"、"DES" → 加密算法
- "http"、"api"、"request" → 网络请求
- "premium"、"vip"、"paid" → 会员检查
- "signature"、"sign" → 签名算法
- "root"、"frida"、"xposed" → 反检测

定位关键代码：

1. 从入口 Activity 开始（`MainActivity.onCreate()`）
2. 检查 Application 子类（`Application.onCreate()` - 初始化逻辑）
3. 搜索字符串常量（右键 → "查找用法"）
4. 分析网络请求（OkHttp、Retrofit、HttpURLConnection）
5. 追踪用户输入处理（`onClick` 回调）

代码导航快捷键：

- Ctrl+H: 查看类层次结构
- Ctrl+F12: 查看当前类的所有方法

## 步骤 2：识别代码模式

### 正常代码

```
// 可读的类名和方法名
public class LoginManager {
 private static final String API_URL = "https://api.example.com/login";

 public boolean login(String username, String password) {
 String encryptedPassword = AESUtil.encrypt(password);
 return ApiClient.post(API_URL, username, encryptedPassword);
 }
}
```

### 混淆代码

```
// ProGuard/R8 混淆
public class a {
 private static final String a = "https://api.example.com/login";

 public boolean a(String str, String str2) {
 String b = b.a(str2); // 字符串常量通常会保留
 return c.a(a, str, b);
 }
}
```

## 步骤 3：分析 Native 库

如果应用包含 `.so` 文件，核心算法通常在这里实现。

## 方法 A：使用 IDA Pro 分析

```
1. 打开 SO 文件
ida64 target_unpacked/lib/arm64-v8a/libnative-lib.so

2. 等待自动分析完成

3. 查看导出函数 (Exports 窗口)
查找 JNI 函数命名模式:
Java_com_example_app_NativeHelper_encrypt
Java_<包名>_<类名>_<方法名>

4. 反编译关键函数 (F5 反编译为伪代码)
```

## 方法 B：使用 Ghidra 分析

```
1. 启动 Ghidra
ghidraRun

2. 新建项目 → 导入文件 → 选择 .so 文件

3. 双击文件 → 自动分析

4. 窗口 → Symbol Tree → Exports
查看导出函数列表

5. 双击函数 → 反编译 (右侧面板显示 C 伪代码)
```

## 方法 C：快速命令行分析

```
查看导出函数
nm -D libnative-lib.so | grep Java
输出示例:
00012340 T Java_com_example_app_Crypto_encrypt
00012680 T Java_com_example_app_Crypto_decrypt
00012a00 T Java_com_example_app_Sign_generate

搜索字符串 (可能找到加密密钥)
strings libnative-lib.so | grep -i "key\|secret\|password"
```

## 步骤 4：创建分析笔记

```
分析目标

- 提取登录 API 签名算法
- 绕过 VIP 会员检查
- 获取加密密钥

已定位的关键类/方法

1. `com.example.app.utils.SignUtil.generateSign(Map params)` - 签名生成
2. `com.example.app.user.UserManager.isPremium()` - 会员检查
3. Native: `Java_com_example_app_Crypto_encrypt` - 加密函数

Hook 策略

- Hook `generateSign()` 查看参数和返回值
- Hook `isPremium()` 强制返回 true
- Hook native 函数获取加密密钥

预期挑战

- 签名算法可能在 native 层
- 可能有 Frida 检测
- 网络请求可能有 SSL pinning
```

## 阶段二产出

- 理解应用的核心功能和业务逻辑
- 定位关键类、方法和 native 函数
- 识别使用的加密/签名算法
- 确定动态分析的 hook 点清单
- 识别潜在的反调试/反 hook 机制

## 阶段三：动态分析（2-4 小时）

目标：在运行时观察实际行为，验证静态分析结论，绕过保护机制。

## 步骤 1：设置 Frida 环境

```
1. 启动 Frida Server (在设备上)
adb push frida-server /data/local/tmp/
adb shell "chmod 755 /data/local/tmp/frida-server"
adb shell "/data/local/tmp/frida-server &"

2. 验证连接 (在 PC 上)
frida-ps -U
应该看到设备上运行的进程列表

3. 测试 hook
frida -U -f com.example.app --no-pause
进入交互式控制台
```

## 步骤 2：编写 Hook 脚本

基于静态分析结果，编写 Frida 脚本。

### 示例 1：Hook 会员检查

```
// hook_premium.js - 绕过 VIP 检查

Java.perform(function () {
 console.log("[+] 开始 hook 会员检查...");

 var UserManager = Java.use("com.example.app.user.UserManager");

 // Hook isPremium 方法
 UserManager.isPremium.implementation = function () {
 console.log("[+] isPremium() 被调用");

 // 调用原始方法查看真实结果
 var realResult = this.isPremium();
 console.log(" 真实返回值: " + realResult);

 // 强制返回 true
 console.log(" 修改返回值: true");
 return true;
 };

 console.log("[+] Hook 完成");
});
```

## 示例 2：提取签名算法

```
// hook_sign.js - 提取签名算法

Java.perform(function () {
 var SignUtil = Java.use("com.example.app.utils.SignUtil");

 SignUtil.generateSign.implementation = function (params) {
 console.log("\n[SIGN] generateSign() 被调用");
 console.log(" 参数类型: " + params.$className);

 // 如果是 Map, 遍历打印
 if (params.$className === "java.util.HashMap") {
 var HashMap = Java.use("java.util.HashMap");
 var entrySet = params.entrySet();
 var iterator = entrySet.iterator();

 console.log(" 参数内容:");
 while (iterator.hasNext()) {
 var entry = iterator.next();
 var key = entry.getKey();
 var value = entry.getValue();
 console.log(" " + key + " = " + value);
 }
 }

 // 调用原始方法
 var result = this.generateSign(params);

 console.log(" 签名结果: " + result);
 console.log(" 签名长度: " + result.length);

 // 打印调用栈
 console.log(" 调用栈:");
 console.log(
 Java.use("android.util.Log").getStackTraceString(
 Java.use("java.lang.Exception").$new()
)
);
 };

 return result;
};

console.log("[+] 签名 hook 完成");
});
```

### 示例 3: Hook Native 函数

```
// hook_native.js – Hook native 加密函数

var encryptAddr = Module.findExportByName(
 "libnative-lib.so",
 "Java_com_example_app_Crypto_encrypt"
);

if (encryptAddr) {
 console.log("[+] 找到 encrypt 函数: " + encryptAddr);

 Interceptor.attach(encryptAddr, {
 onEnter: function (args) {
 console.log("\n[NATIVE] encrypt() 被调用");
 console.log(" JNIEnv*: " + args[0]);
 console.log(" jobject: " + args[1]);

 // 第 3 个参数通常是 jstring (输入数据)
 try {
 var env = Java.vm.getEnv();
 var inputStr = env.getStringUtfChars(args[2], null);
 var input = inputStr.readCString();
 console.log(" 输入: " + input);
 env.releaseStringUtfChars(args[2], inputStr);
 } catch (e) {
 console.log(" 输入: [无法读取]");
 }
 },
 onLeave: function (retval) {
 // 返回值也是 jstring (密文)
 try {
 var env = Java.vm.getEnv();
 var outputStr = env.getStringUtfChars(retval, null);
 var output = outputStr.readCString();
 console.log(" 输出: " + output);
 env.releaseStringUtfChars(retval, outputStr);
 } catch (e) {
 console.log(" 输出: " + retval);
 }
 },
 });
}

console.log("[+] Native hook 完成");
} else {
 console.log("[+] 未找到 encrypt 函数");
}
```

### 步骤 3：绕过反调试检测

```
// bypass_all.js - 综合绕过脚本

Java.perform(function () {
 console.log("[+] 加载反检测模块...");

 // 1. 绕过 Frida 端口检测
 var connect = Module.findExportByName("libc.so", "connect");
 Interceptor.attach(connect, {
 onEnter: function (args) {
 var sockaddr = ptr(args[1]);
 var port = (sockaddr.add(2).readU8() << 8) | sockaddr.add(3).readU8();
 if (port === 27042 || port === 27043) {
 console.log("[检测] 拦截了 Frida 端口扫描");
 sockaddr.add(2).writeU8(0xff);
 }
 },
 });
});

// 2. 绕过 TracerPid 检测
var fgets = Module.findExportByName("libc.so", "fgets");
Interceptor.attach(fgets, {
 onLeave: function (retval) {
 if (retval && !retval.isNull()) {
 var line = retval.readCString();
 if (line && line.includes("TracerPid:")) {
 retval.writeUtf8String("TracerPid:\t0\n");
 console.log("[检测] 修改 TracerPid 为 0");
 }
 }
 },
});
};

// 3. 绕过字符串检测
var strstr = Module.findExportByName("libc.so", "strstr");
Interceptor.attach(strstr, {
 onLeave: function (retval) {
 if (this.needle && this.needle.toLowerCase().includes("frida")) {
 retval.replace(ptr(0));
 console.log("[检测] 隐藏 Frida 字符串");
 }
 },
 onEnter: function (args) {
 this.needle = args[1].readCString();
 },
});
};

console.log("[+] 反检测模块加载完成");
});
```

## 步骤 4：运行分析

```
组合使用多个脚本
frida -U -f com.example.app \
-l bypass_all.js \
-l hook_premium.js \
--no-pause
```

## 步骤 5：网络流量分析

```
1. 配置代理
adb shell settings put global http_proxy 192.168.1.100:8080

2. 启动应用并绕过 SSL pinning
frida -U -f com.example.app -l bypass_ssl_pinning.js --no-pause

3. 在 Burp Suite 中查看流量
```

## Hook OkHttp 请求

```
// hook_okhttp.js
Java.perform(function () {
 var RealInterceptorChain = Java.use(
 "okhttp3.internal.http.RealInterceptorChain"
);

 RealInterceptorChain.proceed.implementation = function (request) {
 console.log(
 "\n[HTTP] " + request.method() + " " + request.url().toString()
);
 // 打印请求头
 var headers = request.headers();
 for (var i = 0; i < headers.size(); i++) {
 console.log(" " + headers.name(i) + ": " + headers.value(i));
 }

 var response = this.proceed(request);

 console.log("[RESP] Code: " + response.code());

 return response;
 };
});
```

## 阶段三产出

- 成功绕过会员检查、反调试等限制
- 完整的网络请求/响应日志
- 准备好重打包的修改点

## 阶段四：代码修改与重打包（30-60 分钟）

目标：对应用进行永久性修改，实现持久化的功能改变。

### 步骤 1：修改 Smali 代码

基于动态分析结果，在 Smali 层面进行修改。

#### 示例 1：绕过会员检查

原始 Java 代码（Jadx 反编译）：

```
public boolean isPremium() {
 return this.userInfo.vipStatus == 1;
}
```

原始 Smali 代码：

```
.method public isPremium()Z
.locals 2

读取 userInfo.vipStatus
.iget-object v0, p0, Lcom/example/app/user/UserManager;-->userInfo:Lcom/example/app/
model/UserInfo;
.iget v0, v0, Lcom/example/app/model/UserInfo;-->vipStatus:I

比较是否等于 1
const/4 v1, 0x1
if-ne v0, v1, :cond_0

如果相等, 返回 true
const/4 v0, 0x1
return v0

如果不相等, 返回 false
:cond_0
const/4 v0, 0x0
return v0
.end method
```

修改后 Smali 代码:

```
.method public isPremium()Z
.locals 1

直接返回 true, 跳过所有检查
const/4 v0, 0x1
return v0
.end method
```

示例 2: 移除广告显示

原始 Smali 代码:

```
.method private showAd()V
 .locals 1

 # 检查是否不是 VIP
 invoke-virtual {p0}, Lcom/example/app/MainActivity;.>isPremium()Z
 move-result v0

 # 如果不是 VIP, 显示广告
 if-nez v0, :cond_0
 invoke-direct {p0}, Lcom/example/app/MainActivity;.>loadAdView()V

 :cond_0
 return-void
.end method
```

修改后 Smali 代码:

```
.method private showAd()V
 .locals 0

 # 直接返回, 不执行任何操作
 return-void
.end method
```

## 步骤 2: 重新打包 APK

```
1. 使用 Apktool 重打包
apktool b target_unpacked -o modified.apk

输出:
I: Using Apktool 2.x.x
I: Checking whether sources has changed...
I: Smali folder: smali
I: Smali folder: smali_classes2
I: Copying raw resources...
I: Copying libs... (/lib)
I: Copying assets... (/assets)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk into: modified.apk

2. 检查生成的 APK
ls -lh modified.apk
-rw-r--r-- 1 user user 8.5M modified.apk
```

## 步骤 3: 签名 APK

```
1. 生成签名密钥（只需执行一次）
keytool -genkey -v \
 -keystore my-release-key.keystore \
 -alias my-key-alias \
 -keyalg RSA \
 -keysize 2048 \
 -validity 10000

提示输入密码和信息:
Enter keystore password: [输入密码]
Re-enter new password: [再次输入]
What is your first and last name? [随意填写]
...

2. 签名 APK
apksigner sign \
 --ks my-release-key.keystore \
 --ks-key-alias my-key-alias \
 --out signed.apk \
 modified.apk

提示输入 keystore 密码
输出: signed.apk

3. 验证签名
apksigner verify signed.apk
输出: Verifies
表示签名成功
```

## 使用 uber-apk-signer (更简单)

```
java -jar uber-apk-signer.jar --apks modified.apk
输出: modified-aligned-debugSigned.apk
```

## 步骤 4：安装测试

```
1. 卸载原应用（如果存在）
adb uninstall com.example.app

2. 安装修改后的 APK
adb install signed.apk

如果遇到签名冲突：
adb install -r signed.apk (替换安装)

3. 启动应用
adb shell am start -n com.example.app/.MainActivity

4. 查看日志验证修改
adb logcat | grep "example.app"
```

验证清单：

- 修改的功能生效（例如 VIP 权限解锁）
- 没有崩溃或异常行为
- 网络功能正常（如果修改了签名相关代码）

## 常见问题排查

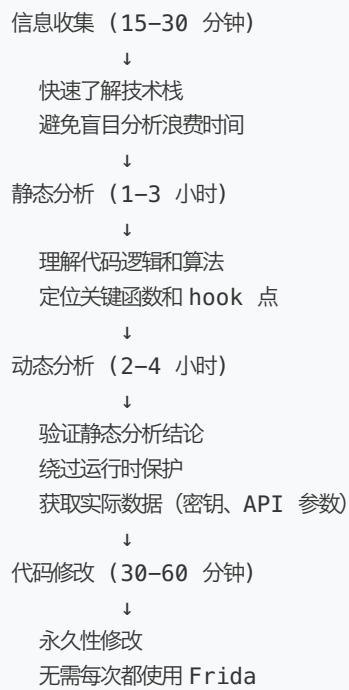
```
查看崩溃日志
adb logcat | grep "AndroidRuntime"

常见错误：
1. "INSTALL_PARSE_FAILED_NO_CERTIFICATES"
→ 签名失败，重新签名

2. "INSTALL_FAILED_UPDATE_INCOMPATIBLE"
→ 签名不匹配，先卸载原应用

3. 应用崩溃
→ 查看 logcat，可能是 Smali 语法错误
```

## 工作流程总结



## 静态 vs 动态分析对比

场景	静态分析	动态分析	推荐
绕过混淆	困难	可行	动态
获取密钥	难（可能硬编码）	易（运行时）	动态
修改代码	精确	不持久	静态
时间成本	高（有混淆时）	中等	结合使用

## 常见问题

### 问题 1：Apktool 解包失败

错误信息：`brut.androlib.AndrolibException: Could not decode arsc file`

可能原因：

1. APK 使用了资源混淆 (AndResGuard)
2. APK 已损坏
3. Apktool 版本过旧

解决方案：

```
1. 更新 Apktool 到最新版本
下载: https://ibotpeaches.github.io/Apktool/

2. 使用 -r 参数跳过资源解码
apktool d target.apk -r -o target_unpacked
-r: 不解码资源文件 (resources.arsc)

3. 使用 --only-main-classes 仅解码主 DEX
apktool d target.apk --only-main-classes -o target_unpacked

4. 如果只需要 Smali, 直接使用 dex2jar + jd-gui
d2j-dex2jar target.apk
生成 target-dex2jar.jar, 用 JD-GUI 打开
```

### 问题 2：代码混淆严重无法阅读

解决方案：

```
1. 优先使用动态分析
混淆的代码在运行时行为不变
使用 Frida 直接 hook，观察参数和返回值

2. 利用字符串常量定位
字符串通常不会被混淆
在 JADX 中搜索关键字字符串，反向定位代码

3. 重命名类/方法 (Jadx 支持)
右键类名 → Rename
根据功能手动重命名为有意义的名称

4. 使用符号还原工具
如果有 mapping.txt (混淆映射文件)
可以使用工具还原符号
```

### 问题 3: Frida 无法连接应用

可能原因:

1. 应用未运行
2. 包名错误
3. Frida Server 未启动

解决方案:

```
1. 检查 Frida Server 是否运行
adb shell ps | grep frida-server
如果没有输出, 需要启动 Frida Server

2. 确认应用正在运行
adb shell ps | grep com.example.app
或
frida-ps -U | grep example

3. 使用正确的包名
查看已安装应用的包名
adb shell pm list packages | grep example

4. 使用 spawn 模式 (自动启动应用)
frida -U -f com.example.app -l script.js --no-pause
-f: spawn 模式, 会自动启动应用

5. 检查设备连接
adb devices
应显示: device (不是 offline 或 unauthorized)
```

## 问题 4：重打包后应用崩溃

可能原因：

1. Smali 语法错误
2. 修改破坏了类结构
3. 缺少依赖

解决方案：

```
1. 查看详细崩溃日志
adb logcat -c # 清空日志
adb logcat | grep -E "AndroidRuntime|FATAL"

2. 验证 Smali 语法
用 Apktool 重新反编译修改后的 APK
apktool d signed.apk -o verify_unpacked
查看是否有错误提示

3. 回滚修改, 逐步测试
先修改单个方法, 通过后再修改其他

4. 使用 baksmali/smali 验证
baksmali d modified.apk -o smali_test
smali a smali_test -o test.dex
如果验证通过, 说明 Smali 语法正确

5. 检查方法签名是否正确
确保修改的方法签名与接口/父类匹配
```

## 相关资源

### 相关 Recipe

Recipe	说明
<a href="#">Android 应用网络流量分析</a>	详细的网络流量分析步骤
<a href="#">绕过应用对 Frida 的检测</a>	反调试绕过
<a href="#">脱壳和分析加固的 Android 应用</a>	处理加壳应用
<a href="#">Frida 常用脚本速查</a>	现成的脚本模板

### 工具深入

- [• Frida 使用指南 - 完整的 Frida 使用手册](#)

- [Ghidra 使用指南](#) - Native 代码分析
- [IDA Pro 使用指南](#) - 专业逆向工程工具

## 案例研究

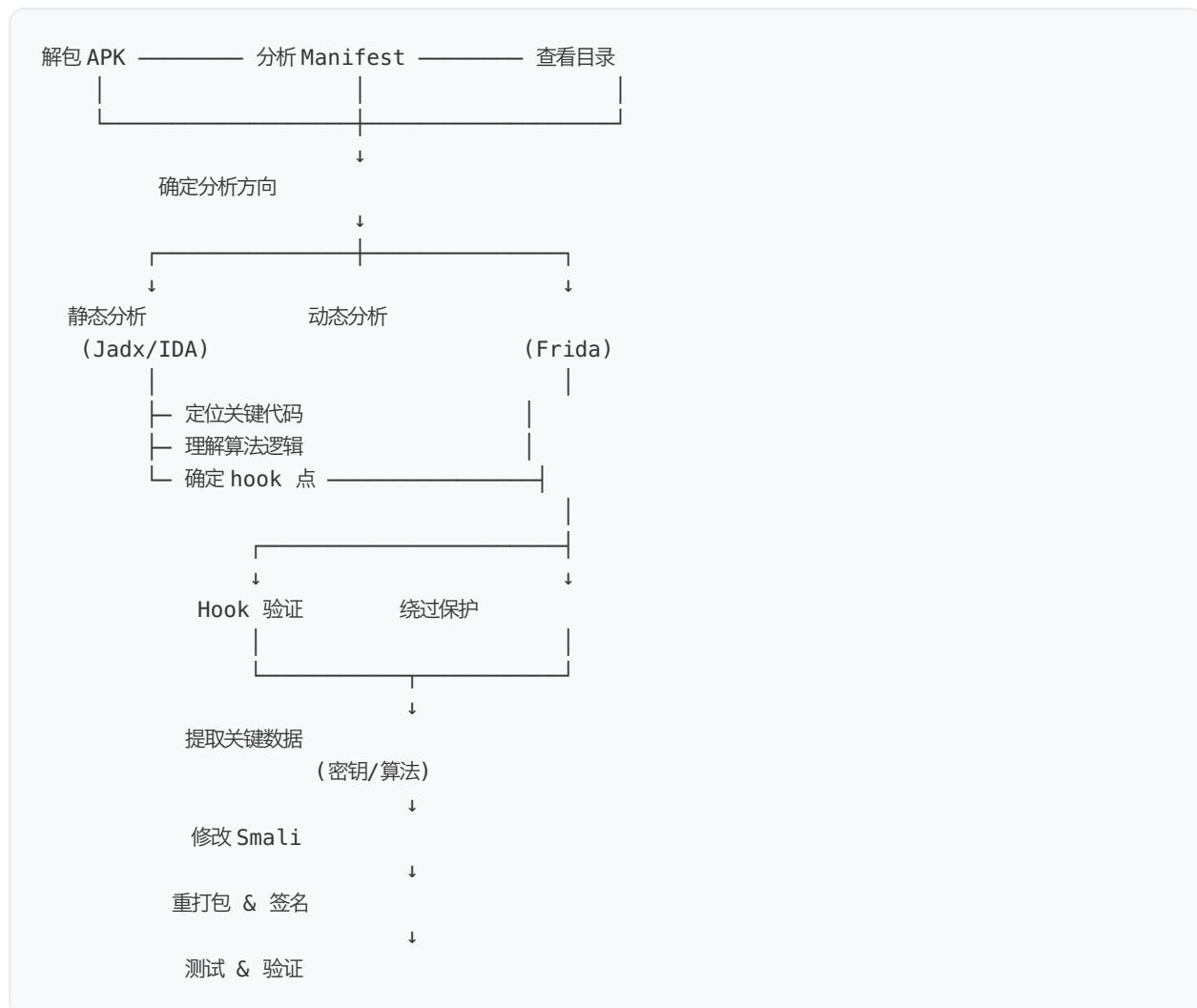
- [案例：音乐应用分析](#) - 完整工作流程实践
- [案例：应用加密分析](#)

## 参考资料

- [APK 文件结构详解](#)
  - [Smali 语法参考](#)
  - [Android 组件详解](#)
-

## 速查手册

### 工作流程快速地图



## 常用命令速查

操作	工具	命令
解包	Apktool	<code>apktool d app.apk -o unpacked</code>
反编译	Jadx	<code>jadx-gui app.apk</code>
Native 分析	IDA/ Ghidra	直接打开 <code>.so</code> 文件
动态分析	Frida	<code>frida -U -f &lt;pkg&gt; -l script.js --no-pause</code>
重打包	Apktool	<code>apktool b unpacked -o modified.apk</code>
签名	apksigner	<code>apksigner sign --ks key.keystore --out signed.apk modified.apk</code>
安装	ADB	<code>adb install signed.apk</code>

## 常用快捷操作

```
1. 一键提取 APK 脚本 (保存为 get-apk.sh)
#!/bin/bash
PKG=$1
APK_PATH=$(adb shell pm path $PKG | cut -d: -f2 | tr -d '\r')
adb pull $APK_PATH ./${PKG}.apk
echo "[+] 已保存: ${PKG}.apk"

使用: ./get-apk.sh com.example.app

2. 一键解包 + 反编译
apktool d app.apk && jadx-gui app.apk &

3. 快速查看Manifest
apktool d -s app.apk -o temp && cat temp/AndroidManifest.xml

4. 自动签名脚本 (保存为 sign-apk.sh)
#!/bin/bash
PKG=$1
java -jar uber-apk-signer.jar --apks $PKG
echo "[+] 签名 APK 已创建"

5. Frida 快速hook (交互模式)
frida -U com.example.app
进入后执行:
Java.perform(function() {
var cls = Java.use("com.example.Class");
cls.method.implementation = function() { return true; };
});


```

## 决策树



---

## [R17] 静态分析深入

# R17: 使用静态分析定位 Android App 的关键逻辑

## 问题场景

你遇到了什么问题？

- 你想找到 App 的加密/签名算法，但代码太多不知道从哪里开始
- 你想理解 App 的完整业务逻辑，包括所有分支和边界条件
- 你想寻找安全漏洞，比如硬编码密钥、逻辑缺陷
- 你想在没有运行环境的情况下分析 APK
- 你想进行批量自动化分析

本配方教你：系统性地使用静态分析工具（jad, IDA Pro, Ghidra）快速定位关键代码、追踪数据流、识别加密算法。

核心理念：

静态分析：不运行代码，看清全局

- 静态分析能看到所有代码路径（包括未触发的分支）
- 适合理解完整算法和寻找漏洞
- 先动态获取线索，再静态深入分析
- 交替迭代：动态发现 → 静态验证 → 动态测试

预计用时：40-90 分钟

## 工具清单

### 必需工具

- jadx-gui - Java/Smali 反编译
- IDA Pro / Ghidra - Native 层分析
- 文本编辑器 - 记录分析笔记

### 可选工具

- Binary Ninja - 可视化 CFG
- FindCrypt (IDA 插件) - 识别加密算法
- YARA - 模式匹配
- angr - 符号执行 (高级)

## 前置条件

### 确认清单

```
1. jadx-gui 已安装
jadx-gui --version

2. IDA Pro or Ghidra 可用
IDA Pro: 商业软件
Ghidra: 免费, 下载自 https://ghidra-sre.org/

3. APK 文件已解压
unzip app.apk -d app_unzipped
```

## 静态 vs 动态：何时选择什么？

你的目标	推荐起点	理由
快速提取结果（如加密参数）	动态优先	直接 Hook 拿结果，不必理解全部逻辑
理解完整算法（如协议逆向）	静态优先	需要看清所有分支和边界条件
寻找漏洞	静态优先	需要覆盖所有代码路径，包括错误处理
对抗混淆/加壳	动态优先	静态分析可能完全失效，先动态脱壳
批量自动化	静态优先	动态分析需要运行环境，静态可离线

最佳实践：

1. 先动态获取线索 - 用 Frida 快速定位关键函数
2. 再静态深入分析 - 有了“地图”后更有方向性
3. 交替迭代 - 动态发现的疑点用静态验证

## 解决方案

### 第 1 步：确定分析目标（5 分钟）

明确你想找什么：

- API 签名算法
- 加密密钥位置
- 网络协议逻辑
- 特定功能实现（如登录、支付）
- 安全漏洞

示例：假设目标是找到 API 请求的签名逻辑

## 第 2 步：从字符串入手（10 分钟）

最有效的起点：搜索关键字符串

### 2.1 jadx-gui 字符串搜索

在 jadx-gui 中使用 `Ctrl+Shift+F` 搜索以下关键词：

```
md5
sha
hmac
key
secret
encrypt
sign
```

找到可疑代码后分析调用关系：

```
HashMap<String, String> params = new HashMap<>();
params.put("sign", generateSign(data));
```

## 第 3 步：交叉引用分析（10 分钟）

### 3.1 查找函数调用者

在 jadx-gui 中：

1. 右键点击 `generateSign` 函数
2. 选择 "Find Usage" 或按 `X`
3. 查看所有调用点

在 IDA Pro 中：

1. 光标移到函数名
2. 按 `X` 键

### 3. 查看 Xrefs to (被谁调用) 和 Xrefs from (调用了谁)

#### 3.2 向上追溯调用链

```
RequestBuilder.buildParams()
↓
SignUtils.generateSign() ← 目标函数
```

分析签名函数的实现：

```
public static String generateSign(Map<String, String> params) {
 // Step 1: Sort parameters
 TreeMap<String, String> sortedParams = new TreeMap<>(params);

 // Step 2: Concatenate string
 StringBuilder sb = new StringBuilder();
 for (Map.Entry<String, String> entry : sortedParams.entrySet()) {
 sb.append(entry.getKey()).append("=").append(entry.getValue()).append("&");
 }
 sb.append("key=").append(SECRET_KEY);

 // Step 3: Calculate MD5
 return MD5.encode(sb.toString());
}
```

## 第 4 步：数据流分析 (15 分钟)

目标：追踪 SECRET\_KEY 的来源

#### 4.1 查找变量定义

在 jadx 中：

1. 点击 SECRET\_KEY
2. Ctrl+Click 跳转到定义

可能的情况：

情况 1：硬编码（最简单）

```
private static final String SECRET_KEY = "abc123def456";
```

直接拿到密钥！

情况 2: 从配置文件读取

```
static {
 try {
 Properties props = new Properties();
 props.load(context.getAssets().open("config.properties"));
 SECRET_KEY = props.getProperty("api.secret");
 } catch (IOException e) {
 SECRET_KEY = null;
 }
}
```

需要检查配置文件：

```
cat app_unzipped/assets/config.properties
```

情况 3: 从 Native 层获取

```
static {
 System.loadLibrary("native-lib");
 SECRET_KEY = getKeyFromNative();
}

private static native String getKeyFromNative();
```

需要分析 SO 文件。

## 4.2 在 Ghidra 中追踪数据流

在 Ghidra 反编译窗口：

1. 双击变量名
2. 所有使用该变量的地方会高亮显示
3. 追踪变量在函数内的流动

## 第 5 步：Native 层分析（20 分钟）

如果关键逻辑在 SO 文件中。

### 5.1 定位 Native 函数

在 jadx 中找到 JNI 声明：

```
public native String encrypt(String plaintext);
```

找到对应的 SO 文件：

```
ls app_unzipped/lib/arm64-v8a/
libnative-lib.so

用 IDA Pro 打开
```

### 5.2 在 IDA Pro 中定位函数

使用函数窗口搜索：

1. View → Open Subviews → Functions
2. 搜索 `Java_com_example` 前缀
3. 双击跳转到函数

或使用 Exports 窗口：

1. View → Open Subviews → Exports
2. 搜索函数名
3. 双击跳转

### 5.3 分析函数逻辑

示例反编译代码（IDA/Ghidra）：

```
jstring Java_com_example_CryptoUtils_encrypt(JNIEnv *env, jobject obj, jstring plaintext) {
 const char *plain = (*env)->GetStringUTFChars(env, plaintext, 0);

 // AES encryption
 unsigned char key[16] = {0x12, 0x34, 0x56, 0x78, ...};
 unsigned char iv[16] = {0x00, 0x00, 0x00, 0x00, ...};

 unsigned char *encrypted = aes_encrypt(plain, key, iv);

 jstring result = (*env)->NewStringUTF(env, encrypted);
 (*env)->ReleaseStringUTFChars(env, plaintext, plain);

 return result;
}
```

提取信息：

- 密钥: {0x12, 0x34, 0x56, 0x78, ...}
- IV: {0x00, 0x00, ...}

## 第 6 步：识别加密算法（10 分钟）

### 6.1 使用 FindCrypt 插件 (IDA Pro)

安装：

```
下载
git clone https://github.com/polymorf/findcrypt-yara.git

复制到 IDA 插件目录
cp findcrypt3.py $IDA_PATH/plugins/
```

运行插件后会自动标记识别到的加密常量。

### 6.2 手动识别

常见加密算法特征：

算法	特征常量（十六进制）
AES	63 7C 77 7B F2 6B 6F C5 (S-Box)
MD5	67 45 23 01 EF CD AB 89 (初始化向量)
SHA-1	67 45 23 01 EF CD AB 89 98 BA DC FE
SHA-256	428A2F98 71374491 B5C0FBCF
DES	固定的 S-Box 和 P-Box 表

在 IDA 中搜索：使用 `Alt+B`（二进制搜索）查找特征字节。

## 第 7 步：控制流图分析（10 分钟）

用途：理解复杂函数的逻辑结构

### 7.1 查看 CFG

IDA Pro：按 `Space` 键切换到图形视图。

示例登录流程图：

```

[检查用户名] --No--> [返回错误 1]
 ↓ Yes
[检查密码长度] --No--> [返回错误 2]
 ↓ Yes
[加密密码]
 ↓
[发送网络请求]
 ↓
[解析响应] --Failed--> [返回错误 3]
 ↓ Success
[保存 Token]
 ↓
[返回成功]

```

## 7.2 交叉引用图

Xrefs to (被谁调用) :

```
LoginActivity.login()
↓
SignUtils.generateSign()
```

Xrefs from (调用了谁) :

```
SignUtils.generateSign()
↓
→ MD5.encode()
→ Base64.encode()
```

## 常见问题

### 问题 1: jadx 反编译失败

症状: 打开 APK 后显示错误或代码不完整

解决:

#### 1. 尝试不同版本的 jadx

```
使用最新版
wget https://github.com/skylot/jadx/releases/download/v1.4.7/jadx-1.4.7.zip
```

#### 1. 调整 jadx 设置

- 禁用 "Deobfuscation"
- 禁用 "Inline methods"

#### 2. 查看 Smali 代码

```
使用 apktool
apktool d app.apk -o app_smali
```

## 问题 2: 找不到关键字符串

可能原因:

1. 字符串被加密/混淆
  - 在运行时动态解密
  - 解决: 用 Frida Hook 查看运行时字符串
2. 字符串在 Native 层

```
在 SO 文件中搜索
strings libnative-lib.so | grep "sign"
```

1. 字符串被拆分

```
// Code might be
String key = "sec" + "ret" + "key";
```

## 问题 3: IDA Pro 没有自动识别函数

症状: 打开 SO 文件后只看到数据, 没有函数

解决:

1. 手动创建函数
  - 按 'P' 键创建函数
  - 按 'C' 键转换为代码
2. 使用自动分析
  - 勾选 "Create Functions"

- 勾选 "Analyze Code"

### 3. 检查是否被混淆

- OLLVM 控制流平坦化
  - 参考: [OLLVM 反混淆](#)

## 问题 4: 代码太复杂看不懂

策略:

### 1. 重命名变量和函数

IDA: 按 'N' 键重命名

Ghidra: 右键 → Rename

```
// Original
String a = b(c, d);

// After renaming
String encryptedData = encrypt(plaintext, key);
```

### 1. 添加注释

IDA: 按 ';' 键添加注释

Ghidra: 右键 → Set Comment

### 1. 分段理解

- 一次只分析一个功能
- 画流程图记录逻辑

## 问题 5: 如何验证静态分析结果?

### 方法 1: 使用 CyberChef

访问 <https://gchq.github.io/CyberChef/> 测试加密算法。

## 方法 2: Python 重现

```
import hashlib

def generate_sign(params, secret_key):
 # 从静态分析复制的逻辑
 sorted_params = sorted(params.items())
 sign_str = '&'.join([f'{k}={v}' for k, v in sorted_params])
 sign_str += f'&key={secret_key}'
 return hashlib.md5(sign_str.encode()).hexdigest()

测试
params = {'user': 'test', 'timestamp': '1234567890'}
secret = 'abc123'
print(generate_sign(params, secret))
```

## 方法 3: Frida 对比

```
Java.perform(function () {
 var SignUtils = Java.use("com.example.SignUtils");
 var HashMap = Java.use("java.util.HashMap");

 var params = HashMap.$new();
 params.put("user", "test");
 params.put("timestamp", "1234567890");

 var sign = SignUtils.generateSign(params);
 console.log("[*] 签名结果:", sign);
});
```

## 延伸阅读

### 相关 Recipes

- [逆向工程工作流 - 完整的分析流程](#)
- [密码学分析 - 加密算法识别](#)
- [OLLVM 反混淆 - 处理混淆代码](#)

## 工具深入

- [IDA Pro 使用指南](#)
- [Ghidra 使用指南](#)

## 在线资源

资源	链接
IDA Pro 教程	<a href="https://www.hex-rays.com/products/ida/support/tutorials/">https://www.hex-rays.com/products/ida/support/tutorials/</a>
Ghidra 官方手册	<a href="https://ghidra-sre.org/docs/">https://ghidra-sre.org/docs/</a>
FindCrypt 插件	<a href="https://github.com/polymorf/findcrypt-yara">https://github.com/polymorf/findcrypt-yara</a>

## 理论基础

- [ARM 汇编基础](#)
  - [DEX 文件格式](#)
  - [ELF 文件格式](#)
-

## 快速参考

### jadx 快捷键

快捷键	功能
Ctrl+Shift+F	全局搜索
Ctrl+F	当前文件搜索
X	查找用法 (Xrefs)
Ctrl+Click	跳转到定义
Alt+←	后退
F5	重新反编译

## IDA Pro 快捷键

快捷键	功能
X	交叉引用
N	重命名
;	添加注释
Space	切换图形/文本视图
G	跳转到地址
P	创建函数
C	转换为代码
A	转换为字符串
Shift+F12	查看字符串
Alt+T	文本搜索
Alt+B	二进制搜索

## 分析流程模板

1. 分析目标: \_\_\_\_\_
2. 入口点: \_\_\_\_\_
3. 关键函数: \_\_\_\_\_
4. 调用链:  
\_\_\_\_\_ → \_\_\_\_\_ → \_\_\_\_\_
5. 关键变量:
  - 名称: \_\_\_\_\_
  - 类型: \_\_\_\_\_
  - 来源: \_\_\_\_\_
6. 算法识别: \_\_\_\_\_
7. 验证结果: \_\_\_\_\_
8. 下一步: \_\_\_\_\_

成功定位关键逻辑了吗？现在你可以理解 App 的核心算法了！

下一步推荐：[动态分析深入](#)（验证和测试你的发现）

---

## [R18] 动态分析深入

# R18: 使用动态分析验证和探索 Android App 的运行时行为

## 问题场景

你遇到了什么问题？

- 你已经静态分析找到了目标函数，现在想验证它的实际输入输出
- 你想捕获运行时才生成的数据（如动态密钥、签名结果）
- 你想绕过 SSL Pinning / 反调试 / Root 检测
- 你想主动调用函数测试不同参数的效果
- 你想追踪代码执行路径，看看哪些函数被调用了

本配方教你：系统性地使用 Frida、调试器、追踪工具来验证静态分析结果、获取运行时数据、绕过保护机制。

核心理念：

### 动态分析：让代码说话

- 动态分析验证静态分析的假设
- 获取只在运行时存在的数据
- 主动探索程序的内部状态
- Hook → Debug → Trace 三种武器各有用途

预计用时：30-90 分钟

## 工具清单

### 必需工具

- Frida - 动态插桩框架
- Android 设备（已 Root）或模拟器
- Python 3.7+ - 运行 Frida 脚本

### 可选工具

- IDA Pro Remote Debugger - Native 层调试
- objection - Frida 的交互式工具
- Burp Suite - 网络抓包
- GDB - GNU 调试器

## 前置条件

### 确认清单

```
1. Frida 正常运行
frida-ps -U

2. Python 环境
python3 --version

3. 目标 App 已安装
adb shell pm list packages | grep <app_name>
```

## Hook vs Debug vs Trace: 何时用什么?

场景	推荐工具	理由
想知道某个函数的输入输出	Frida Hook	最快速, 不中断程序流
想理解复杂算法的每一步细节	IDA/GDB 调试器	可以单步执行, 查看每个变量
想知道程序执行了哪些代码路径	Stalker/Trace	全自动记录, 无需设断点
想绕过某个检测 (如 SSL Pinning)	Frida Hook	直接替换函数返回值
想找到某个字符串是在哪里生成的	内存断点 + 调试器	在写入时中断
想分析反调试机制	Frida + 调试器组合	先用 Frida 禁用, 再用调试器分析

经验法则:

- 能用 Hook 解决的, 别用调试器 (效率问题)
- 需要理解逻辑的, 必须用调试器 (深度问题)
- 需要全局视野的, 用追踪 (覆盖率问题)

## 解决方案

### 第 1 步: 验证静态分析结果 (15 分钟)

假设静态分析发现了签名函数: `SignUtils.generateSign()`

#### 1.1 Hook 函数查看输入输出

基础 Hook 脚本 `verify_sign.js`:

```
Java.perform(function () {
 console.log("[*] Start Hook SignUtils.generateSign");

 var SignUtils = Java.use("com.example.SignUtils");

 SignUtils.generateSign.implementation = function (params) {
 console.log("\n[*] generateSign is called!");
 console.log(" InputParameter:");

 // Print HashMap
 var iterator = params.entrySet().iterator();
 while (iterator.hasNext()) {
 var entry = iterator.next();
 console.log(" " + entry.getKey() + " = " + entry.getValue());
 }

 // Call original function
 var result = this.generateSign(params);

 console.log(" ReturnValue: " + result);
 console.log("");

 return result;
 };

 console.log("[*] Hook install completed");
});
```

运行结果示例：

```
[*] generateSign is called!
 InputParameter:
 user = test123
 timestamp = 1701234567
 action = login
 ReturnValue: a1b2c3d4e5f6g7h8i9j0
```

## 第 2 步：处理重载方法（10 分钟）

### 2.1 列出所有重载

```
Java.perform(function () {
 var CryptoUtil = Java.use("com.example.CryptoUtil");

 // List all encryption methods
 console.log("[*] encrypt Method overloads:");
 CryptoUtil.encrypt.overloads.forEach(function (overload) {
 console.log(" " + overload);
 });
});
```

输出示例：

```
encrypt(java.lang.String)
encrypt(java.lang.String, java.lang.String)
encrypt([B)
```

## 2.2 Hook 特定重载

```
Java.perform(function () {
 var CryptoUtil = Java.use("com.example.CryptoUtil");

 // Hook the second overloaded version
 CryptoUtil.encrypt.overload(
 "java.lang.String",
 "java.lang.String"
).implementation = function (data, key) {
 console.log("[*] encrypt(String, String) is called");
 console.log(" Data:", data);
 console.log(" Key:", key);

 var result = this.encrypt(data, key);
 console.log(" Result:", result);

 return result;
 };

 // Hook the third overloaded version
 CryptoUtil.encrypt.overload("[B").implementation = function (bytes) {
 console.log("[*] encrypt(byte[]) is called");
 console.log(" BytesLength:", bytes.length);

 var result = this.encrypt(bytes);
 console.log(" ResultLength:", result.length);

 return result;
 };
});
```

## 第3步：主动调用函数（15分钟）

### 3.1 创建新实例

```
Java.perform(function () {
 var CryptoUtil = Java.use("com.example.CryptoUtil");

 // Constructor accessible
 try {
 var instance = CryptoUtil.$new(); // Call no-arg constructor
 var result = instance.encrypt("Hello World", "mykey");
 console.log("[*] MainCallResult:", result);
 } catch (e) {
 console.log("[-] No way to create instance:", e);
 }
});
```

### 3.2 使用已有实例

```
Java.perform(function () {
 Java.choose("com.example.CryptoUtil", {
 onMatch: function (instance) {
 console.log("[+] Instance:", instance);

 // Actively call
 var encrypted = instance.encrypt("test data");
 console.log("[*] EncryptResult:", encrypted);

 var decrypted = instance.decrypt(encrypted);
 console.log("[*] DecryptResult:", decrypted);
 },
 onComplete: function () {
 console.log("[*] Search completed");
 },
 });
});
```

### 3.3 调用静态方法

```
Java.perform(function () {
 var SignUtils = Java.use("com.example.SignUtils");
 var HashMap = Java.use("java.util.HashMap");

 // Create Parameter
 var params = HashMap.$new();
 params.put("user", "testuser");
 params.put("timestamp", String(Date.now()));

 // Actively call static method
 var sign = SignUtils.generateSign(params);
 console.log("[*] GenerateSignature:", sign);
});
```

## 第 4 步：绕过安全检测（15 分钟）

### 4.1 绕过 Root 检测

```
Java.perform(function () {
 var RootDetector = Java.use("com.example.security.RootDetector");

 RootDetector.isRooted.implementation = function () {
 console.log("[*] Root Detection is Bypassed");
 return false; // Force return false (not rooted)
 };

 RootDetector.isXposedInstalled.implementation = function () {
 console.log("[*] Xposed Detection is Bypassed");
 return false;
 };
});
```

---

## 4.2 绕过 SSL Pinning

```
Java.perform(function () {
 // Hook OkHttp 3
 try {
 var CertificatePinner = Java.use("okhttp3.CertificatePinner");
 CertificatePinner.check.overload(
 "java.lang.String",
 "java.util.List"
).implementation = function (hostname, peerCertificates) {
 console.log("[*] Bypass OkHttp3 SSL Pinning:", hostname);
 return; // Return directly, skip validation
 };
 console.log("[+] OkHttp3 SSL Pinning Bypassed");
 } catch (e) {
 console.log("[-] OkHttp3 not found");
 }

 // Hook TrustManager
 try {
 var X509TrustManager = Java.use("javax.net.ssl.X509TrustManager");
 var SSLContext = Java.use("javax.net.ssl.SSLContext");

 var TrustManager = Java.registerClass({
 name: "com.example.TrustManager",
 implements: [X509TrustManager],
 methods: {
 checkClientTrusted: function (chain, authType) {},
 checkServerTrusted: function (chain, authType) {},
 getAcceptedIssuers: function () {
 return [];
 },
 },
 });
 });

 var TrustManagers = [TrustManager.$new()];
 var SSLContext_init = SSLContext.init.overload(
 "[Ljavax.net.ssl.KeyManager;",
 "[Ljavax.net.ssl.TrustManager;",
 "java.security.SecureRandom"
);
 SSLContext_init.implementation = function (
 keyManager,
 trustManager,
 secureRandom
) {
 console.log("[*] Replacing TrustManager");
 SSLContext_init.call(this, keyManager, TrustManagers, secureRandom);
 };
 console.log("[+] TrustManager Bypassed");
} catch (e) {
 console.log("[-] TrustManager bypass failed:", e);
}
```

```
 console.log("[*] SSL Pinning Bypass Config Completed");
 });
}
```

## 第 5 步：使用 RPC 远程调用（20 分钟）

### 5.1 定义 RPC 函数

rpc\_example.js:

```
rpc.exports = {
 // Export Function: Generate Signature
 generateSign: function (params) {
 var result = null;

 Java.perform(function () {
 var SignUtils = Java.use("com.example.SignUtils");
 var HashMap = Java.use("java.util.HashMap");

 var map = HashMap.$new();
 for (var key in params) {
 map.put(key, params[key]);
 }

 result = SignUtils.generateSign(map);
 });

 return result;
 },

 // Export Function: Encrypt Data
 encrypt: function (plaintext, key) {
 var result = null;

 Java.perform(function () {
 var CryptoUtil = Java.use("com.example.CryptoUtil");
 result = CryptoUtil.encrypt(plaintext, key);
 });

 return result;
 },
};
```

## 5.2 Python 调用 RPC

rpc\_caller.py:

```
import frida
import sys
import time

def on_message(message, data):
 print(f"[*] Message: {message}")

Connect to device
device = frida.get_usb_device()

Attach to process
pid = device.spawn(['com.example.app'])
session = device.attach(pid)

Load Script
with open('rpc_example.js', 'r') as f:
 script = session.create_script(f.read())

script.on('message', on_message)
script.load()

device.resume(pid)

Wait for Script Initialize
time.sleep(2)

Call RPC Function
params = {
 'user': 'testuser',
 'timestamp': '1701234567',
 'action': 'login'
}

sign = script.exports_sync.generate_sign(params)
print(f"[+] Generated Signature: {sign}")

encrypted = script.exports_sync.encrypt('Hello World', 'mykey')
print(f"[+] Encrypt Result: {encrypted}")

Keep running
sys.stdin.read()
```

## 第 6 步：使用调试器进行深度分析（20 分钟）

### 6.1 IDA Pro 远程调试 Native 代码

准备：

```
1. Push android_server to device
adb push android_server64 /data/local/tmp/
adb shell chmod 755 /data/local/tmp/android_server64

2. Run with root permission
adb shell su -c "/data/local/tmp/android_server64"

3. Port forward
adb forward tcp:23946 tcp:23946
```

连接步骤：

1. IDA Pro 中选择 Debugger → Remote ARM Linux/Android debugger
2. 配置连接参数：
  - Hostname: localhost
  - Port: 23946
3. Debugger → Attach to Process → 选择目标 App
4. 在目标函数处设置断点 (F2)
5. 触发 App 中的操作，断点命中

### 6.2 使用 GDB 调试

```
Attach to process
adb shell
su
ps | grep <app_name>
找到 PID, 如 12345

gdbserver :5039 --attach 12345
```

在主机上连接：

```
arm-linux-androideabi-gdb
```

```
In GDB
(gdb) target remote :5039
(gdb) continue
```

### 6.3 调试器快捷键

IDA Pro:

快捷键	功能
F2	设置/取消断点
F9	运行/继续
F7	单步进入 (Step Into)
F8	单步跳过 (Step Over)
Ctrl+F7	执行到返回 (Run to Return)

GDB:

命令	功能
<code>break &lt;addr&gt;</code>	设置断点
<code>continue</code>	继续执行
<code>step</code>	单步进入
<code>next</code>	单步跳过
<code>finish</code>	执行到返回
<code>info registers</code>	查看寄存器
<code>x/10x \$sp</code>	查看栈内容

## 第 7 步：使用 Stalker 追踪代码覆盖率（15 分钟）

Frida Stalker 可以记录线程执行的所有指令。

## 7.1 基础 Stalker 示例

```
// Stalker trace function execution
Interceptor.attach(
 Module.findExportByName("libnative.so", "Java_com_example_Native_encrypt"),
 {
 onEnter: function (args) {
 console.log("[*] Start Tracking...");

 Stalker.follow(Process.getCurrentThreadId(), {
 events: {
 call: true, // Record Function Calls
 ret: false,
 exec: false,
 },
 onReceive: function (events) {
 console.log("[*] Captured", events.length, "events");

 // Parse events
 var calls = Stalker.parse(events, {
 annotate: true,
 stringify: false,
 });

 calls.forEach(function (call) {
 console.log(" Call:", call);
 });
 },
 });
 },
 onLeave: function (retval) {
 Stalker.unfollow(Process.getCurrentThreadId());
 Stalker.flush();
 console.log("[*] Tracking Ended");
 },
 }
);
```

## 7.2 只追踪特定模块

```

var base = Module.findBaseAddress("libnative.so");
var size = Process.findModuleByName("libnative.so").size;

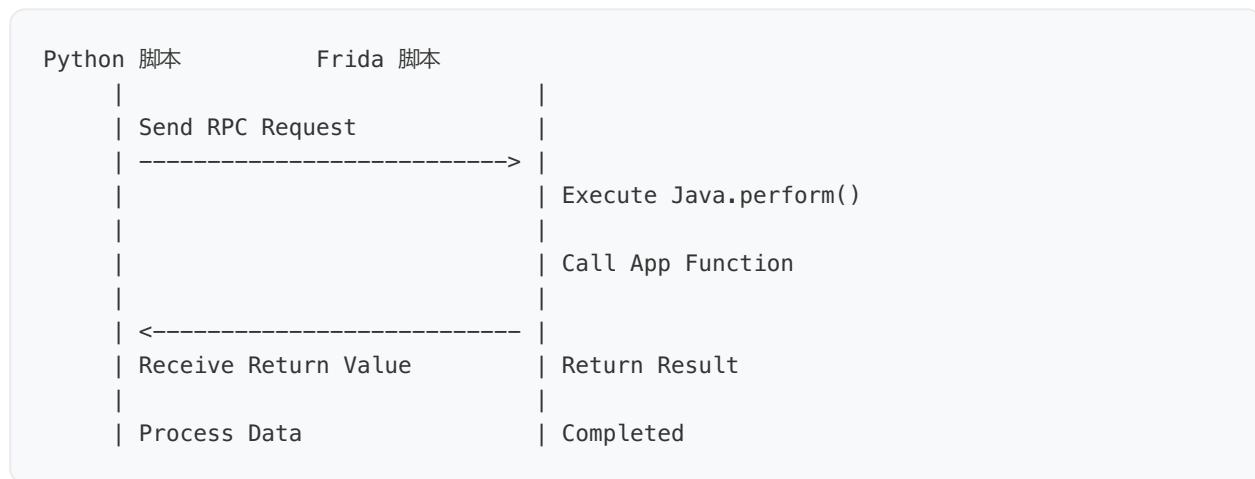
Stalker.follow(Process.getCurrentThreadId(), {
 transform: function (iterator) {
 var instruction = iterator.next();
 do {
 // Only record instructions within libnative.so
 if (
 instruction.address.compare(base) >= 0 &&
 instruction.address.compare(base.add(size)) < 0
) {
 iterator.keep();
 }
 instruction = iterator.next();
 } while (instruction !== null);
 },
});

```

## 工具对比总结

工具	最佳场景	优点	缺点
Frida Hook	快速获取 I/O	不中断流程	只看单点
	修改返回值	易于自动化	不知道细节
调试器	理解算法逻辑	完全控制	速度慢
(IDA/GDB)	单步跟踪	看所有变量	需要手动操作
Stalker	代码覆盖率	全自动	性能开销大
	追踪执行路径	无需断点	输出量巨大

## RPC 调用流程



## 常见问题

### 问题 1: Hook 没有触发

检查清单:

#### 1. 确认类/方法名正确

```
// List all classes
Java.enumerateLoadedClasses({
 onMatch: function (className) {
 if (className.includes("SignUtils")) {
 console.log(className);
 }
 },
 onComplete: function () {},
});
```

#### 1. 确认方法被调用

- 在 App 中触发相关操作
- 检查是否有其他代码路径

#### 2. 检查混淆

```
// If class name is obfuscated as a.b.c, use obfuscated name
var SignUtils = Java.use("a.b.c");
```

## 问题 2: RPC 调用超时

症状: `script.exports_sync.func()` 一直等待

解决:

```
Use async call
def on_rpc_message(result, error):
 if error:
 print(f"[-] Error: {error}")
 else:
 print(f"[+] Result: {result}")

script.exports.func(params, on_rpc_message)

Or add timeout
result = script.exports_sync.func(params, timeout=10)
```

## 问题 3: Frida 被检测

常见检测方式:

### 1. 检查端口

```
// App Code
Socket socket = new Socket("127.0.0.1", 27042); // Frida default port
```

绕过: 修改 Frida Server 端口

```
frida-server -l 0.0.0.0:8888
```

### 1. 检查 maps 文件

```
BufferedReader reader = new BufferedReader(new FileReader("/proc/self/maps"));
if (line.contains("frida")) {
 System.exit(0);
}
```

绕过：使用魔改版 Frida

```
strongR-frida
wget https://github.com/hluwa/strongR-frida-android/releases/download/xxx/frida-server
```

## 问题 4: 调试器无法附加

症状: IDA Pro 显示 "Cannot attach to process"

解决:

### 1. 检查 SELinux

```
adb shell getenforce
If is Enforcing
adb shell setenforce 0
```

### 1. 确认进程存在

```
adb shell ps | grep <app_name>
确认 PID 正确
```

### 1. 检查 ptrace 权限

```
adb shell
su
echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

## 问题 5: Stalker 导致 App 卡死

症状: 启用 Stalker 后 App 卡死

---

优化:

1. 只追踪关键模块 (见第 7.2 步)

2. 减少事件类型

```
events: {
 call: true, // Only record function calls
 ret: false, // Don't record returns
 exec: false // Don't record every instruction
}
```

1. 使用 transform 过滤

```
transform: function(iterator) {
 // Skip code we don't care about
}
```

## 延伸阅读

### 相关 Recipes

- [静态分析深入](#) - 先静态找到目标
- [Frida 常用脚本](#) - Hook 脚本模板
- [Frida 反调试](#) - 绕过检测
- [SSL Pinning 绕过](#) - 抓包必备

### 工具深入

- [Frida 完整指南](#)
- [IDA Pro 调试](#)

## 在线资源

资源	链接
Frida 官方文档	<a href="https://frida.re/docs/">https://frida.re/docs/</a>
Frida Codeshare	<a href="https://codeshare.frida.re/">https://codeshare.frida.re/</a>
Frida Handbook	<a href="https://learnfrida.info/">https://learnfrida.info/</a>

## 理论基础

- [ARM 汇编](#) - 理解 Native 调试
- [ART 运行时](#) - 理解 Java Hook

## 快速参考

### Frida Hook 模板

Hook Java 方法:

```
Java.perform(function () {
 var ClassName = Java.use("com.example.ClassName");

 ClassName.methodName.implementation = function (arg1, arg2) {
 console.log("[*] methodName called");
 console.log(" arg1:", arg1);
 console.log(" arg2:", arg2);

 var result = this.methodName(arg1, arg2);
 console.log(" result:", result);

 return result;
 };
});
```

## Hook Native 方法:

```
Interceptor.attach(Module.findExportByName("libnative.so", "native_func"), {
 onEnter: function (args) {
 console.log("[*] native_func called");
 console.log(" arg0:", args[0]);
 console.log(" arg1:", args[1]);
 },
 onLeave: function (retval) {
 console.log(" retval:", retval);
 },
});
```

## RPC 导出模板:

```
rpc.exports = {
 callMethod: function (className, methodName, args) {
 var result = null;
 Java.perform(function () {
 var Class = Java.use(className);
 result = Class[methodName].apply(Class, args);
 });
 return result;
 },
};
```

## IDA Pro 调试快捷键

快捷键	功能
F2	设置断点
F9	继续执行
F7	单步进入
F8	单步跳过
Ctrl+F7	执行到返回

## GDB 常用命令

命令	功能
<code>break &lt;addr&gt;</code>	设置断点
<code>continue</code>	继续执行
<code>step</code>	单步进入
<code>next</code>	单步跳过
<code>finish</code>	执行到返回
<code>info registers</code>	查看寄存器
<code>x/10x \$sp</code>	查看栈内容

成功验证分析结果了吗？ 现在你可以获取运行时数据了！

下一步推荐：[Frida 常用脚本（更多实用脚本模板）](#)

---

## [R19] OLLVM 反混淆

# R19: OLLVM 反混淆

OLLVM (Obfuscator-LLVM) 是一个著名的开源代码混淆框架，它在 LLVM 编译器 IR (中间表示) 层面进行操作。这使其能够与具体语言无关，并对代码应用复杂的、难以逆向的转换。

!!! warning "场景导入：当你遇到 OLLVM" 打开 IDA，反编译一个函数，结果看到：

- 一个巨大的 `switch-case` 循环，有几十甚至上百个 case 分支
- 每个 case 里只有几行代码，然后又跳回 switch
- 到处都是看起来有用实际无用的 `if` 判断
- 简单的加法被替换成了 `a = b - (-c)` 这样的怪异表达式

你的第一反应可能是：这是什么鬼？

恭喜，你遇到了 OLLVM 控制流平坦化 (FLA) + 虚假控制流 (BCF) + 指令替换 (SUB) 的"三件套"。这是目前 Android Native 层最常见的商业级混淆方案。

关键问题：面对这种混淆，是选择"硬看"代码，还是有更聪明的办法？

本文档涵盖了常见的 OLLVM 混淆通道 (pass) 及其分析和逆向策略。

## 核心混淆技术

OLLVM 的主要优势在于其三种核心混淆技术：

1. 控制流平坦化 (`-fla`)：该技术会彻底平坦化一个函数的控制流。它通过将所有基本块放入一个单一的、巨大的分发器循环 ("主分发器") 中来隐藏原始的程序流程。一个状态变量用于控制下一个要执行的代码块。逆向此技术需要重建原始的控制流图 (CFG)。

2. 虚假控制流 (**-bcf**): 该技术在代码中插入无效的条件分支和不透明谓词。这些分支被设计为静态分析难以解析，但在运行时，它们总是会得出相同的结果。这会给控制流图增加大量的噪声。

3. 指令替换 (**-sub**): 这是最简单的混淆方式。它将标准的二进制运算符（如 `add`, `sub`, `and`, `or`）替换为功能上等价但更复杂的指令序列。例如，`a = b + c` 可能会变成 `a = b - (-c)`。

## 分析与反混淆策略

!!! question "思考：静态分析 vs 动态分析，哪个更有效？" 面对 OLLVM 混淆，有两种完全不同的思路：

静态分析：

- ✓ 优势：能看到所有可能的执行路径，包括错误处理分支
- ✗ 劣势：需要对抗大量的虚假分支，分析工作量巨大
- 适用场景：你需要理解完整的算法逻辑，或者寻找漏洞

动态分析：

- ✓ 优势：直接记录真实执行路径，绕过所有虚假分支
- ✗ 劣势：只能看到当前输入下的执行路径，可能遗漏关键分支
- 适用场景：你只想提取算法结果（如加密签名），不关心内部逻辑

实战建议：

1. 先用动态分析（Frida Stalker / Unidbg trace）快速获取“真实”的执行流
2. 再用静态分析验证和补充动态分析遗漏的部分
3. 如果目标是自动化（如算法还原），考虑符号执行（Angr）

## 1. 静态分析

- CFG 重建: 对于控制流平坦化, 关键是识别状态变量和分发器。通过符号执行或模式匹配分发器逻辑, 可以确定每个真实基本块的后继, 从而重建原始图。
- 不透明谓词求解: Z3 或其他 SMT 求解器等工具可用于自动证明虚假控制流中的条件是不变的。这使得分析师能够识别并移除无效的代码路径。
- 模式匹配: 对于指令替换, 可以识别并替换简单的模式。例如, 像  
`x = rdtsc(); y = x & 1; if (y == 0) ...` 这样的序列是一个常见的虚假谓词。

## 2. 动态分析

- 使用 Frida/Unidbg 进行追踪: 动态追踪非常有效。通过使用 Frida 的 Stalker 或 Unidbg 的追踪功能, 可以记录运行时执行的基本块的确切顺序。这可以绕过所有的控制流混淆, 为你提供"真实"的执行路径。
- 符号执行: 像 Angr 这样的引擎可用于探索程序状态。符号执行可以自动求解路径约束, 从而有效地反混淆控制流并简化不透明谓词。这个过程可能很慢, 但功能非常强大。

## 3. 自动化工具

- d-obfuscator: 一个基于 Python 的工具, 使用符号执行 (通过 Angr) 来反混淆 OLLVM。
- QB-DI: 一个基于 QBDI 动态插桩框架的交互式反混淆工具。
- Triton: 一个动态二进制分析框架, 可以通过编写脚本来执行污点分析和符号执行。

## [R20] VMP 虚拟机分析

# R20: VMP 分析

VMP (VMProtect 的简称) 是一种非常强大的软件保护解决方案，它使用虚拟化（一个“虚拟机”）来保护代码。受保护的代码不再执行原生 CPU 指令，而是被转换成一种自定义的字节码，只有特定的、嵌入的虚拟机才能解释执行。

分析受 VMP 保护的代码是逆向工程中最具挑战性的任务之一。

## 核心概念

1. 虚拟机 (VM): VMP 的核心。它包括：

- 解释器循环: 读取字节码并执行相应的处理程序 (handler)。
- 处理程序 (Handlers): 实现每个自定义字节码指令逻辑的小段原生代码（例如，虚拟加法、虚拟跳转）。
- 虚拟寄存器: VM 使用自己的一套虚拟寄存器来存储数据，这些虚拟寄存器会映射到原生寄存器或内存位置。
- VM 上下文: 一个存储 VM 所有状态的结构，包括虚拟寄存器、标志位和虚拟指令指针 (VIP)。

1. 字节码: 原始的原生代码被翻译成一个专为嵌入式 VM 设计的自定义、非标准的指令集。

2. 突变 (Mutation): 为了让分析更加困难，VMP 会为每个新版本的受保护软件突变 VM 的处理程序和字节码。这意味着一个“虚拟加法”指令的处理程序每次都会看起来不一样。

## 分析策略

直接对 VMP 进行去虚拟化极其困难，通常也不是主要目标。重点通常是理解某个特定受保护函数的逻辑。

### 1. 识别 VM 及其组件

- 入口点分析: 第一步是定位"VM 入口", 即程序从执行原生代码切换到 VM 解释器循环的地方。其特征通常是一系列的 PUSH 指令, 用以保存原生上下文, 并最终通过一个 JMP/ CALL 指令进入解释器。
- 处理程序追踪: 一旦进入 VM, 追踪执行流将揭示不同处理程序的地址。通过分析每个处理程序对 VM 上下文做了什么, 就可以开始勾勒出虚拟指令集。例如, 一个读取两个虚拟寄存器、将它们相加并存储结果的处理程序很可能就是"虚拟加法"。

### 2. 动态分析与追踪

- 指令追踪: 这是最有效的方法。目标是构建一个正在执行的虚拟指令的追踪记录。这需要:
  1. 在解释器循环的开始处 (或每个处理程序上) 设置断点。
  2. 在每一步, 转储虚拟机的状态 (虚拟寄存器、VIP)。
  3. 记录处理程序执行的原生操作。
- Frida/DBI 工具: 像 Frida、Pin 或 QBDI 这样的动态二进制插桩 (DBI) 框架至关重要。它们允许你编写追踪过程的脚本, 在解释器的每一步自动记录 VM 状态。

### 3. 重建逻辑

- 语义重建: 收集到虚拟指令的追踪记录后, 就可以开始重建高层逻辑。例如, 一个"加载"、"相加"、"存储"的虚拟指令序列可以被翻译回类似 C 的表达式 `var = a + b;`。
- 提升到更高级别的 IR: 完整的去虚拟化工作的最终目标是将自定义字节码"提升"到一个标准的中间表示, 如 LLVM IR 或 REIL。这让你能够使用标准的编译器。这是一个非常复杂的、研究级别的问题。

## 4. 关键要点

不要试图反编译 VM 本身。相反，追踪 VM 的执行来理解它在做什么。它所调用的处理程序的序列才是你需要分析的真正逻辑。

---

## [R21] JS 混淆器分析

# R21: JavaScript Obfuscator (OB 混淆)

## 分析

`javascript-obfuscator` 是一个非常流行和强大的开源工具，用于混淆和保护 JavaScript 代码。它的混淆产物通常被称为“OB 混淆”。OB 混淆通过多种手段的组合，使得代码难以阅读、理解和调试。

### 核心混淆技术

OB 混淆主要依赖于以下几种关键技术：

#### 1. 字符串混淆 (String Concealing)

- 字符串数组：将代码中所有的字符串（特别是敏感信息）提取出来，放入一个或多个巨大的数组中。
- 编码与加密：这些字符串通常会使用 Base64、RC4 或其他自定义算法进行编码或加密。
- 解码函数：提供一个或多个解码函数。在代码执行时，通过调用 `decoder("0x1")` 这样的形式来获取原始字符串。
- 数组乱序与自愈：为了防止静态分析，字符串数组的顺序会在脚本执行初期被动态打乱，解码函数也会随之调整，增加了静态还原的难度。

#### 2. 控制流平坦化 (Control Flow Flattening)

这是 OB 混淆最核心、最复杂的特征之一。

- 状态机转换：将原始代码块（如函数体内的语句）分割成多个小的代码片段，并放入一个巨大的 `while` 循环中的 `switch` 结构里。

- 状态变量：用一个状态变量（程序计数器）来控制 `switch` 的执行顺序。每个 `case` 执行完毕后，会更新状态变量，决定下一个要执行的 `case`。
- 逻辑打断：原始线性的代码逻辑被完全打乱，变成了在一个巨大的循环中无序跳转，使得人工跟踪变得极其困难。

### 3. 代码转换与编码 (Code Transformation)

- 变量名混淆 (Identifier Mangling)：将有意义的变量名、函数名和属性名替换成无意义的短字符，如 `_0xabc123`。
- 数字常量替换：将代码中的数字常量（如 `123`）替换成十六进制字符串表达式（如 `0x7b`），或者更复杂的表达式，增加阅读难度。
- 代理函数 (Proxy Functions)：将简单的二元运算（如 `a + b`）或对象属性访问（`obj.prop`）替换成对一个代理函数的调用，例如 `add(a, b)`。这使得批量替换和模式识别变得更加困难。
- 死代码注入 (Dead Code Injection)：在代码中插入一些永远不会被执行的、但看起来很复杂的逻辑分支（通常与不透明谓词结合），用来迷惑分析者。

### 4. 反调试与反格式化 (Anti-Debugging)

- `debugger` 语句：在代码中插入 `debugger;` 语句，并且通常会将其包裹在一个无限循环的函数中。当开发者工具打开时，程序会立即暂停并陷入这个循环，阻碍动态调试。
- 函数重绑定：通过 `Function.prototype.constructor` 或 `eval` 来执行代码，使得常规的断点难以命中。
- 反格式化：检测代码是否被美化或格式化，如果发现，则可能进入死循环或执行错误逻辑。

## 分析与反混淆策略

反混淆 OB 代码通常是一个系统性的工程，需要多种工具和技术结合。

## 1. 字符串解密与替换

- 定位解码函数：找到负责从字符串数组中取值并解密的函数。
- 执行解码逻辑：
- 动态执行：在 Node.js 或浏览器环境中，直接调用解码函数，将所有加密的字符串预先解密出来。
- 静态分析：如果解码算法（如 RC4）比较标准，可以编写脚本静态地解密所有字符串。
- 批量替换：编写脚本（通常基于 AST），将代码中所有对解码函数的调用 `decoder("0x1")` 替换成其返回的原始字符串 `"original_string"`。

## 2. 控制流平坦化还原

这是最困难的一步，但也是最有价值的一步。

- AST 分析：使用 Babel 等工具将代码解析成 AST。
- 定位主循环：找到包含 `while(true)` 和 `switch` 的巨大循环体。
- 识别状态变量：找到控制 `switch` 跳转的状态变量和它的初始值。
- 重排代码块：
  1. 提取 `switch` 的 `case` 数组和状态变量的初始跳转顺序。
  2. 根据这个顺序，将每个 `case` 块中的代码按正确的逻辑重新排列。
  3. 移除 `while` 和 `switch` 结构，生成线性的、可读的代码。
- 自动化工具：社区中有一些尝试自动化还原控制流的工具，但由于 OB 混淆变种繁多，通用性有限。

## 3. 其他净化操作

- 常量表达式计算：将 `0x7b` 这样的表达式直接计算成 `123`。

- 代理函数内联：将代理函数的逻辑直接替换回原来的位置，例如将 `add(a, b)` 还原成 `a + b`。
- 死代码移除：通过分析控制流，识别并删除无法访问到的代码块。

## 常用工具

- Babel (核心)：用于解析 (Parse)、转换 (Transform) 和生成 (Generate) JavaScript 代码，是编写反混淆脚本的基础。
- AST Explorer：在线查看 AST 结构，便于编写转换逻辑。
- Node.js / 浏览器控制台：用于动态执行代码片段，特别是解密函数。
- de4js, js-beautify：用于基本的代码格式化和一些简单的反混淆。
- AST-Deobfuscator：一些开源的、基于 AST 的反混淆工具框架，可以作为参考。

---

## [R22] JS 虚拟机保护

# R22: JavaScript VMP 逆向工程

JavaScript VMP（虚拟机保护）是一种高级的代码保护技术，它将原始的 JavaScript 代码转换成一种自定义的、基于虚拟机的字节码。然后，在运行时，一个内置的解释器（或虚拟机）会执行这些字节码。这种方式极大地增加了逆向工程的难度，因为它隐藏了原始的代码逻辑和结构。

## 核心原理

JSVMP 的核心思想是创建一个自定义的指令集和一个对应的虚拟机来执行它。

### 1. 代码转换（编译）：

- 词法分析与解析：将原始 JavaScript 代码解析成抽象语法树（AST）。
- 指令生成：遍历 AST，将代码逻辑转换成自定义的字节码序列。例如，`a + b` 可能会被转换成 `PUSH a; PUSH b; ADD`。
- 虚拟机注入：将实现了解释器、指令调度循环和操作函数的虚拟机（通常用 JavaScript 编写）与生成的字节码打包在一起。

### 1. 运行时执行：

- 虚拟机初始化：设置虚拟机的执行环境，如堆栈（Stack）、程序计数器（PC）和上下文。
- 指令循环（Fetch-Decode-Execute）：
  - Fetch：从字节码数组中获取当前 PC 指向的指令。
  - Decode：解析指令的操作码和操作数。
  - Execute：执行指令对应的操作，例如进行数学运算、操作堆栈、调用函数等。
- 程序结束：当执行完所有字节码后，虚拟机将结果返回或完成操作。

## 常见特征

- 巨大的代码体积：代码中通常包含一个非常大的数组（字节码）和一个庞大的 `switch` 或 `while` 循环（虚拟机解释器）。
- 控制流平坦化：原始的 `if/else`, `for`, `while` 结构被转换成由程序计数器（PC）控制的跳转指令，使得代码逻辑难以跟踪。
- 不透明谓词：引入一些恒为真或恒为假的复杂条件判断，增加静态分析的难度。
- 自定义数据结构：使用自定义的堆栈来存储变量和中间结果，而不是直接使用 JavaScript 的变量。

## 分析与脱壳策略

逆向 JSVMP 是一个复杂的过程，通常需要结合静态和动态分析。

### 1. 静态分析：理解虚拟机

- 定位核心组件：
  - 字节码数组：通常是一个巨大的、包含数字或短字符串的数组。
  - 虚拟机入口：启动整个解释器循环的函数。
  - 分发器（Dispatcher）：通常是一个大的 `switch` 语句或 `while(true)` 循环，根据指令码调用不同的处理函数。
  - 指令处理器（Handlers）：`switch` 中的每个 `case` 或被调用的函数，实现了具体指令的功能。
- 指令集重建：
  - 通过分析每个 Handler 的功能，逐步还原出每个字节码对应的具体操作（如 `ADD`, `SUB`, `JMP` 等）。
  - 为每个操作码（Opcode）添加注释，记录其功能。这是一个非常耗时但至关重要的步骤。

## 2. 动态分析：跟踪与调试

- Hook 关键函数：使用 Frida 或浏览器开发者工具在关键位置（如指令处理器）下断点或插入日志。
- 跟踪 PC 和操作数：记录每次循环的程序计数器（PC）和当前指令的操作数，可以得到完整的执行轨迹（Trace）。
- 监控堆栈变化：观察虚拟机自定义堆栈的压入（push）和弹出（pop）操作，以理解数据流。
- AST 辅助分析：使用工具（如 Babel）将 Handler 的代码解析成 AST，可以更快地理解其功能，甚至自动化地识别指令模式。

## 3. 代码还原与重构

- 编写反编译器：基于已经重建的指令集，编写一个脚本，将字节码序列翻译回更高级、更易读的 JavaScript 代码。这是一个高级步骤，需要对虚拟机有完整的理解。
- 手动逻辑重构：对于不是特别复杂的 VMP，可以通过跟踪执行流程，手动将关键逻辑（如加密算法）用等效的 JavaScript 代码重写出来。

## 常用工具

- 浏览器开发者工具：用于下断点、单步调试和观察变量。
- Frida：用于 Hook 关键函数，实现动态跟踪。
- Babel：用于将 JavaScript 代码解析成 AST，辅助静态分析。
- AST Explorer：一个在线工具，可以方便地查看代码对应的 AST 结构。
- IDA Pro / Ghidra：虽然主要用于原生代码，但它们强大的反汇编和反编译功能可以为理解复杂的 JavaScript 虚拟机逻辑提供借鉴。

---

## [R23] Native 字符串混淆

# R23: Native 层字符串混淆与逆向

在 Android Native 开发 (C/C++) 中，直接将明文字符串硬编码在代码中会带来安全风险。静态分析工具（如 IDA Pro、Ghidra）可以轻易地在二进制文件的 `.rodata` (只读数据) 段中找到这些字符串，从而泄露 API 地址、加密密钥、敏感校验逻辑等信息。因此，开发者通常会采用各种字符串混淆技术来保护这些数据。

## 常见的 Native 字符串混淆技术

### 1. 栈上动态构造 (Stack-based Construction)

这是最简单的一种方法。它避免在数据段中留下完整的字符串，而是在函数运行时，逐个字符地将字符串构造在栈上。

示例代码:

```
void get_secret_url() {
 char url[19];
 url[0] = 'h'; url[1] = 't'; url[2] = 't'; url[3] = 'p';
 url[4] = 's'; url[5] = ':'; url[6] = '/'; url[7] = '/';
 url[8] = 'a'; url[9] = 'p'; url[10] = 'i'; url[11] = '.';
 url[12] = 'e'; url[13] = 'x'; url[14] = 'a'; url[15] = 'm';
 url[16] = 'p'; url[17] = 'l'; url[18] = 'e';
 url[19] = '\0'; // Null terminator
 // ... use url
}
```

示例代码:

```
char* decrypt_string(char* encrypted) {
 char key = 0xAB;
 int len = strlen(encrypted);
 for (int i = 0; i < len; i++) {
 encrypted[i] = encrypted[i] ^ key;
 }
 return encrypted;
}

void use_secret() {
 // "secret_key" Xored with 0xAB
 char encrypted_key[] = { 0xCF, 0xC4, 0xC2, 0xCD, 0xC4, 0xD1, 0xDF, 0xCB, 0xC4, 0xD8,
 0x00 };
 char* secret = decrypt_string(encrypted_key);
 // ... use secret
}
```

优点: 自动化、全局覆盖、对开发者透明。 缺点: 通常需要定制的编译器或工具链。

## 逆向策略

逆向字符串混淆的目标是 批量地、自动化地 将混淆的字符串还原出来。

### 1. 静态分析 (IDA Pro / Ghidra)

静态分析是识别解密例程 (Decryption Routine) 和批量解密的关键。

- 识别解密模式:
- 寻找特征性的循环结构。一个循环遍历内存、执行固定操作 (如 `XOR`) 然后写回, 这通常就是解密函数。
- 在 IDA Pro 中, 这种循环的图形视图非常具有辨识度。
- 定位解密函数:
- 通过交叉引用 (Xrefs) 找到加密数据被哪些函数使用。这些函数很可能就是解密函数。
- 一旦找到一个解密函数, 分析其逻辑 (输入、输出、加密算法)。

- 
- 自动化解密 (IDAPython / Ghidra Script):
    1. 编写脚本: 这是最高效的方法。编写一个脚本来模拟解密逻辑。
    2. 寻找引用: 脚本首先找到所有对解密函数的交叉引用。
    3. 提取参数: 在每个调用点, 脚本向上回溯, 解析传递给解密函数的参数 (加密的数据、密钥等)。
    4. 执行解密: 脚本在内部执行解密算法。
    5. 添加注释: 最后, 将解密后的字符串作为注释添加到 IDA Pro 或 Ghidra 的反汇编代码中。

## 2. 动态分析 (Frida)

当静态分析过于复杂或存在反调试时, 动态分析是最佳选择。

- Hook 解密函数:
  1. 通过初步的静态分析定位到疑似解密函数。
  2. 使用 Frida `Interceptor.attach` 来 Hook 这个函数的入口和出口。
  3. 在 `onEnter` 中, 打印函数的参数 (通常是指向加密数据的指针)。
  4. 在 `onLeave` 中, 打印函数的返回值 (通常是指向已解密的明文字符串的指针)。
  5. 通过运行 App 并触发不同功能, 就可以从日志中收集到大量的明文字符串。
- 内存扫描:
  - 另一种策略是让应用运行一段时间, 然后使用 Frida 脚本或 GameGuardian 等工具扫描整个进程内存, 寻找符合字符串特征 (如 ASCII、UTF-8) 的内存区域。
  - 优点: 无需关心解密逻辑。
  - 缺点: 信息非常嘈杂, 包含大量无用数据; 无法将被加密存储但在运行时未被使用的字符串解密出来。

### 3. 模拟执行 (Emulation)

对于一些独立的、没有太多外部依赖的解密函数，可以使用模拟执行框架（如 **Unicorn Engine**）来解密。

1. 提取代码和数据：从二进制文件中 dump 出解密函数的机器码和需要解密的字节数组。
2. 设置环境：在 Unicorn 中，映射所需的内存区域，将加密数据放入。
3. 模拟执行：设置好初始寄存器状态（如参数指针），然后开始模拟执行解密函数的机器码。
4. 获取结果：执行完毕后，从内存中读回解密后的字符串。

优点：速度比动态分析快，无需运行完整的 App，可绕过反调试。 缺点：环境设置复杂，不适用于有大量系统调用或复杂依赖的函数。

## 自动化

---

## [R24] 自动化与设备农场

# R24: 工程化：自动化与群控系统

---

在虚拟化和容器化解决了"环境"问题之后，自动化和群控系统则负责解决"执行"和"管理"的问题。它们是驱动整个规模化测试和分析流水线运转的核心引擎。

---

## 1. 自动化框架

自动化框架是模拟用户行为、与 App UI 进行交互的工具集。它的核心任务是代替人工，实现对 App 的程序化控制。

### a) 主流框架对比

框架	驱动原理	优点	缺点	适用场景
Appium	WebDriver 协议 -> UIAutomator2/ XCUITest	跨平台 (Android/ iOS)，多语言 支持，生态成 熟，功能强 大。	环境配置复杂， 执行速度相对较 慢，对 App 有 一定的侵入性 (需要安装 WebDriver Agent)。	标准 化的、跨平 台的端到 端 (E2E) 功能测 试。
UIAutomator2 (Python)	Google UIAutomator2	直接与设备通 信，速度快， 稳定，API 简 洁易用。	仅支持 Android，功 能相对 Appium 较少。	纯 Android 平台的快 速自动 化、爬虫 和日常脚 本。
Airtest / Poco	图像识别 + UI 控 件	能够解决无法 获取 UI 控件树 的问题 (如游 戏)，跨引擎 (Unity, Cocos)。	图像识别不稳 定，受分辨率和 UI 变化影响 大，速度慢。	游戏自动 化，黑盒 测试。

### b) 脚本编写的最佳实践

分离 UI 元素与业务逻辑 (Page Object Model): 不要将 UI 元素的定位符 (如 `resource-id`) 硬编码在业务代码中。应该为每个页面创建一个类 (Page Object)，封装该页面的所有元素和操作。当 UI 变化时，你只需要修改对应的 Page Object，而无需改动业务流程代码。

明确的断言: 每个测试用例都应该有明确的成功或失败的判断标准 (断言)。例如，点击登录后，断言"用户名"元素是否出现在下一个页面。

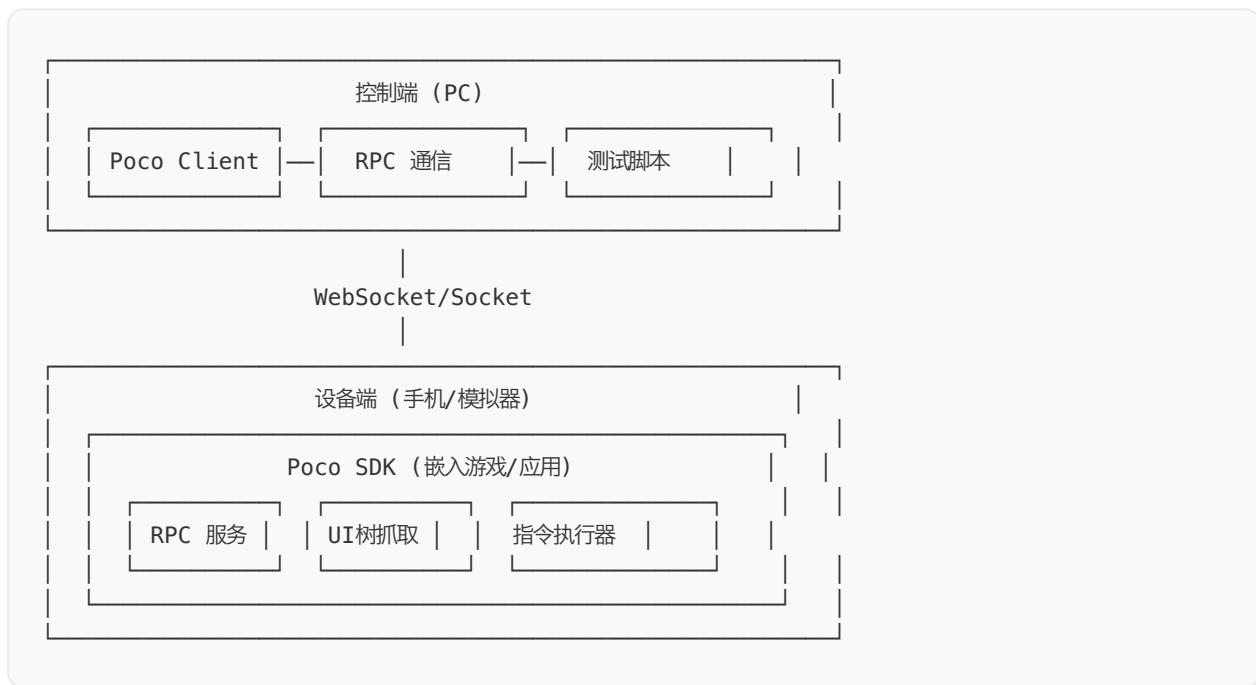
**异常处理与重试:** 网络延迟、系统弹窗等都可能导致自动化失败。在关键步骤加入合理的等待、异常捕获和重试机制，可以大大提高脚本的稳定性。

**日志与报告:** 在脚本的关键节点输出有意义的日志。测试结束后，生成图文并茂的测试报告（如 Allure Report），方便快速定位问题。

### c) Poco 自动化技术深度解析

Poco 是网易推出的 UI 自动化测试框架，专为游戏和复杂应用设计，是 Airtest 项目的核心组件之一。

#### 核心架构与原理



#### 工作流程:

1. SDK 嵌入: 游戏/应用集成 Poco SDK, 启动 RPC 服务
2. 建立连接: 控制端通过 WebSocket 与 SDK 通信
3. UI 树获取: SDK 遍历游戏引擎的 UI 控件, 生成控件树
4. 指令执行: 接收控制端指令, 操作对应 UI 控件

## 5. 结果回传: 将操作结果和状态信息返回控制端

### SDK 集成方式

Unity 引擎集成:

```
// Unity 项目中集成 Poco SDK
using Poco;

public class PocoManager : MonoBehaviour {
 void Start() {
 // 启动 Poco 服务
 var poco = new PocoServiceBuilder()
 .SetPort(5001)
 .SetDebugMode(true)
 .Build();

 poco.Start();
 }
}
```

Cocos2d-x 集成:

```
// Cocos2d-x 项目集成
bool AppDelegate::applicationDidFinishLaunching() {
 // 初始化 Poco 服务
 poco::PocoManager::getInstance()->start();

 return true;
}
```

Android 原生集成:

```
public class MainActivity extends AppCompatActivity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 // 启动 Poco 服务
 Poco.start("poco", 5001);
 }
}
```

## Python 控制端使用

基础连接与操作:

```
from poco.drivers.unity3d import UnityPoco
from poco.drivers.android.uiautomation import AndroidUiautomationPoco

Unity 游戏连接
poco = UnityPoco(('192.168.1.100', 5001))

Android 原生应用连接
poco = AndroidUiautomationPoco(
 use_airtest_input=True,
 screenshot_each_action=False
)

基本操作
poco('Button').click() # 点击按钮
poco('InputField').set_text('test') # 输入文本
poco('ScrollView').swipe('up') # 滑动操作

等待元素出现
poco('LoadingPanel').wait_for_disappearance() # 等待加载完成
```

高级选择器:

```
属性选择
poco(text='确定').click()
poco(name='startBtn', enabled=True).click()
poco(type='Button', visible=True).click()

相对位置选择
poco('username').sibling('password') # 兄弟节点
poco('title').parent() # 父节点

条件过滤
buttons = poco('Button').filter(
 lambda x: x.get_text().startswith('确定')
)
```

游戏专用操作:

```
拖拽操作
poco.drag_to([0.2, 0.2], [0.8, 0.8])

多点触控
poco pinch(in_or_out='in', center=[0.5, 0.5], percent=0.6)

等待游戏状态
def wait_for_battle_end():
 return poco('BattleResult').exists()

poco.wait_for_all(wait_for_battle_end, timeout=60)
```

## 跨引擎适配

```
def get_poco_instance(engine_type, device_info):
 """根据引擎类型创建对应的 Poco 实例"""
 if engine_type == 'unity':
 return UnityPoco(device_info['addr'])
 elif engine_type == 'cocos':
 return CocosJSPoco(device_info['addr'])
 elif engine_type == 'unreal':
 return UE4Poco(device_info['addr'])
 elif engine_type == 'android':
 return AndroidUiautomationPoco()
 else:
 raise ValueError(f"Unsupported engine: {engine_type}")
```

## 连接池管理

```
import queue

class PocoConnectionPool:
 def __init__(self, max_connections=10):
 self.pool = queue.Queue(max_connections)
 self.max_connections = max_connections

 def get_connection(self, addr):
 try:
 return self.pool.get_nowait()
 except queue.Empty:
 return UnityPoco(addr)

 def return_connection(self, conn):
 try:
 self.pool.put_nowait(conn)
 except queue.Full:
 pass # 丢弃多余连接
```

## 批量操作优化

```
def batch_get_elements(poco, selectors):
 """批量获取多个元素，减少 RPC 调用"""
 elements = {}
 for name, selector in selectors.items():
 try:
 elements[name] = poco(selector)
 except:
 elements[name] = None
 return elements

使用示例
ui_elements = batch_get_elements(poco, {
 'start_btn': 'StartButton',
 'settings_btn': 'SettingsButton',
 'exit_btn': 'ExitButton'
})
```

## 稳定性增强

```
from functools import wraps
import time

def retry_on_failure(max_retries=3, delay=1):
 def decorator(func):
 @wraps(func)
 def wrapper(*args, **kwargs):
 for attempt in range(max_retries):
 try:
 return func(*args, **kwargs)
 except Exception as e:
 if attempt == max_retries - 1:
 raise e
 print(f"Attempt {attempt + 1} failed: {e}")
 time.sleep(delay)
 return None
 return wrapper
 return decorator

@retry_on_failure(max_retries=3, delay=2)
def stable_click(poco, selector):
 """稳定的点击操作, 带重试机制"""
 element = poco(selector)
 if element.exists():
 element.click()
 return True
 else:
 raise Exception(f"Element {selector} not found")
```

## 日志与调试

```
import logging
import time
from poco.utils.logger import setup_logger

设置 Poco 日志
setup_logger(level=logging.DEBUG)

自定义操作日志
class PocoLogger:
 @staticmethod
 def log_action(action, element, result=None):
 timestamp = time.strftime("%Y-%m-%d %H:%M:%S")
 print(f"[{timestamp}] {action} on {element}, result: {result}")

 @staticmethod
 def log_screenshot(path):
 print(f"Screenshot saved: {path}")

使用示例
def logged_click(poco, selector):
 try:
 poco(selector).click()
 PocoLogger.log_action("CLICK", selector, "SUCCESS")
 except Exception as e:
 PocoLogger.log_action("CLICK", selector, f"FAILED: {e}")
 # 保存错误时的截图
 screenshot_path = f"error_{int(time.time())}.png"
 poco.snapshot(screenshot_path)
 PocoLogger.log_screenshot(screenshot_path)
 raise
```

## 群控集成示例

```
class DevicePocoController:
 def __init__(self, device_id, poco_port=5001):
 self.device_id = device_id
 self.poco_port = poco_port
 self.poco = None

 def connect(self):
 """连接到设备上的 Poco 服务"""
 device_ip = self.get_device_ip(self.device_id)
 self.poco = UnityPoco((device_ip, self.poco_port))
 return self.poco is not None

 def execute_task(self, task_config):
 """执行自动化任务"""
 try:
 actions = task_config['actions']
 for action in actions:
 self.execute_action(action)
 return {"status": "success", "device_id": self.device_id}
 except Exception as e:
 return {"status": "failed", "error": str(e), "device_id": self.device_id}

 def execute_action(self, action):
 """执行单个操作"""
 action_type = action['type']

 if action_type == 'click':
 self.poco(action['selector']).click()
 elif action_type == 'input':
 self.poco(action['selector']).set_text(action['text'])
 elif action_type == 'wait':
 time.sleep(action['duration'])
 # ... 其他操作类型
```

## 性能优化建议

1. 连接复用: 使用连接池管理 Poco 连接, 避免频繁建立连接
2. 批量操作: 合并多个 UI 操作, 减少 RPC 调用次数
3. 缓存控件: 对于频繁访问的 UI 控件, 缓存其引用
4. 异步操作: 对于耗时操作使用异步模式提高效率
5. 错误处理: 完善的异常捕获和恢复机制
6. 性能监控: 监控 RPC 调用延迟和成功率

## 与其他自动化框架对比

特性	Poco	Appium	UIAutomator2
游戏支持	优秀	不支持	不支持
跨引擎	Unity/Cocos/UE4	仅原生	仅 Android 原生
集成复杂度	中等(需 SDK)	低(无需修改)	低(无需修改)
执行速度	快	中等	快
稳定性	高	中等	高
学习成本	中等	高	低

Poco 特别适合游戏自动化测试、游戏 AI 训练、游戏内容验证等场景，是移动游戏自动化的首选方案。

## 2. 群控系统 (Device Farming)

群控系统是一个将大量物理设备或虚拟设备（模拟器）汇集成一个统一的、可编程的资源池，并对其进行集中化管理、调度和监控的平台。

### a) 核心架构

一个工业级的群控系统通常是微服务架构，包含以下核心组件：

#### 1. API 网关 (API Gateway):

- 作为所有服务的统一入口，负责请求路由、身份认证和速率限制。

#### 2. 设备管理服务 (Device Management Service):

- 维护一个包含所有设备（真机/模拟器）信息的数据库。

- 通过心跳机制实时监控每个设备的状态（空闲、占用、离线、健康状况）。
- 处理设备的接入和注销。

### 3. 任务调度服务 (Task Scheduling Service):

- 接收用户提交的任务（例如，“在 Android 12 上对 App X 执行 Y 测试”）。
- 根据任务要求（设备类型、系统版本等）和预设的调度策略（如优先级、FIFO）从设备管理服务中查询并锁定一个合适的设备。

### 4. 执行代理 (Agent):

- 在每个物理设备或模拟器上运行的一个轻量级代理程序。
- 负责接收并执行来自调度中心的具体指令，如：安装/卸载 APK、启动/停止 Appium、执行 shell 命令、上传/下载文件等。

### 5. 结果收集与报告服务:

- 接收来自 Agent 的实时日志、截图、录屏和测试结果。
- 将结果存入数据库，并在任务结束后生成最终的测试报告。

### 6. Web 管理前端:

- 提供一个可视化的界面，让用户可以实时查看设备列表、远程控制设备（如 STF）、提交任务、查看任务队列和历史报告。

## b) 开源方案与自研

STF (Smartphone Test Farm): 提供了优秀的设备管理和远程控制功能，是许多自研群控系统的基础。但它本身不包含任务调度和报告等功能。

自研: 许多大型公司会基于 STF、Appium、Docker、Kubernetes 等开源技术栈，结合自身的业务需求，搭建自研的群控平台，以实现更灵活的调度逻辑和更深入的业务集成。

---

## 总结

自动化与群控系统是移动端工程化能力的集中体现。它将底层的设备资源、中层的执行脚本和上层的业务需求有机地结合在一起，形成了一个强大的、可扩展的自动化解决方案，是现代 App 开发、测试和安全分析不可或缺的一环。

---

## [R25] 拨号代理池

# R25: 逆向技术：动态住宅 IP 代理池

在高强度的爬虫和逆向分析场景中，请求的 IP 地址是识别和封禁爬虫流量的第一道关卡。相比于易于被识别和屏蔽的数据中心 IP，动态变化的住宅 IP 地址具有极高的伪装度，是绕过反爬虫策略的关键技术。本节将详细介绍动态住宅 IP（拨号代理）的原理及其代理池的搭建方法。

## 1. 动态住宅 IP (拨号代理) 原理

### a) 什么是动态住宅 IP?

- 住宅 IP: 指由互联网服务提供商 (ISP)，如电信、联通、移动，分配给普通家庭用户的 IP 地址。这些 IP 地址属于真实的住宅网络，信誉度最高。
- 动态 IP: 与数据中心固定的静态 IP 不同，住宅宽带通常使用 PPPoE (Point-to-Point Protocol over Ethernet) 协议进行拨号上网。其核心特点是：每断开一次连接再重新拨号，ISP 的 BRAS (宽带远程接入服务器) 就会从其地址池中重新分配一个新的 IP 地址给用户。

利用这一特性，我们可以通过程序自动化地控制 ADSL (或光猫) 进行“断线重拨”，从而在极短的时间内获取一个全新的、干净的、真实的住宅 IP。这就是拨号代理的核心原理。

### b) 优势

- 极高的真实性: IP 来自真实的 ISP 住宅网络，无法被目标服务器通过 IP 库识别为数据中心流量。
- 海量 IP 资源: 一个地区级的 ISP 运营商通常拥有数万到数百万的 IP 地址池。理论上，通过不断重拨，你可以使用该地区的所有 IP。

- 
- 成本可控：相比于按流量计费的商业住宅代理服务，自建拨号代理池（尤其是在拥有物理设备的情况下）的长期成本更低。
- 

## 2. 搭建拨号代理池

搭建一个稳定高效的拨号代理池，需要将物理层的拨号设备、网络层的代理服务和管理层的调度系统结合起来。

### a) 硬件与物理层

1. ADSL/光猫 + 路由器：这是最基础的单元。你需要一个（或多个）办理了宽带业务的 ADSL 猫或光猫，并连接到一个可以被程序控制的路由器。
2. 树莓派/小型 PC：在每个拨号设备旁边，放置一个类似树莓派的廉价小型主机，用于执行拨号和代理服务的指令。
3. 4G/5G 模块（可选）：除了固定宽带，还可以使用 4G/5G 工业模块。通过控制模块的飞行模式切换或重置，同样可以实现 IP 的更换。这种方式灵活性更高，但流量成本也更高。

### b) 软件与网络层

1. 拨号脚本：在树莓派上运行一个脚本，用于控制路由器执行 PPPoE 的断开和重连操作。这通常可以通过 `curl` 或 `ssh` 调用路由器的管理接口来实现。

示例（控制 OpenWrt/LEDE 路由器的脚本）：

```
#!/bin/bash

断开 PPPoE 连接
ssh root@192.168.1.1 'ifdown wan'

等待断开
sleep 3

重新连接
ssh root@192.168.1.1 'ifup wan'

等待连接建立
sleep 5

获取新 IP
NEW_IP=$(ssh root@192.168.1.1 'ifconfig pppoe-wan | grep "inet addr" | cut -d: -f2 |
cut -d" " -f1')

echo "New IP: $NEW_IP"
```

1. 代理服务: 在树莓派上运行一个代理服务程序 (如 Squid, Nginx, Tiny Proxy)。外部请求通过这个代理服务发出, 就会使用当前拨号获得的 IP。

Squid 配置示例 (`squid.conf`):

```
允许所有来源的所有请求
http_access allow all

监听端口
http_port 3128

禁止泄露原始 IP
forwarded_for off
request_header_access Via deny all
request_header_access X-Forwarded-For deny all
```

## c) 管理层与调度

当你有大量的拨号节点时, 一个中心化的管理系统是必不可少的。

1. 中心 API 服务器:

- IP 注册: 每个拨号节点在成功获取新 IP 后, 将 (新IP:端口, 地理位置, ISP) 等信息上报给中心服务器。

- 
- IP 获取: 业务程序（如爬虫）通过调用 API，从中心服务器获取一个当前可用的代理 IP。可以根据需求指定地理位置等条件。
  - IP 续期与心跳: 拨号节点需要定期向中心服务器发送心跳，证明自己仍然在线。如果心跳超时，服务器就将该 IP 从可用池中移除。

## 2. IP 池管理策略:

- 可用性检测: 中心服务器定期主动检测池中代理的连通性，剔除失效的 IP。
- IP 轮换: 当一个 IP 被封禁或使用次数过多时，业务程序可以调用 API 请求中心服务器通知对应的拨号节点执行“换 IP”操作。
- 并发控制: 管理每个代理 IP 当前的并发请求数，避免因过度使用而被封禁。

#### d) 整体架构图

```
graph TD
 subgraph "业务服务器"
 A[爬虫/业务应用] --> B{中心 API Server}
 end

 B -- 获取代理 --> A
 B -- 管理/调度 --> C1
 B -- 管理/调度 --> C2
 B -- 管理/调度 --> C3

 subgraph "拨号节点 1 (上海电信)"
 C1[树莓派] --> D1[代理服务 Squid]
 C1 -- 控制重拨 --> E1[路由器/光猫]
 E1 -- PPPoE --> F1[(ISP 网络)]
 end

 subgraph "拨号节点 2 (北京联通)"
 C2[树莓派] --> D2[代理服务 Squid]
 C2 -- 控制重拨 --> E2[路由器/光猫]
 E2 -- PPPoE --> F2[(ISP 网络)]
 end

 subgraph "拨号节点N (深圳移动)"
 C3[树莓派] --> D3[代理服务 Squid]
 C3 -- 控制重拨 --> E3[路由器/光猫]
 E3 -- PPPoE --> F3[(ISP 网络)]
 end

 C1 -- 上报 IP --> B
 C2 -- 上报 IP --> B
 C3 -- 上报 IP --> B
```

### 3. 节点管理脚本示例

拨号节点客户端

```
import requests
import subprocess
import time
import socket

class DialUpNode:
 def __init__(self, api_server, node_id, location, isp):
 self.api_server = api_server
 self.node_id = node_id
 self.location = location
 self.isp = isp
 self.current_ip = None
 self.proxy_port = 3128

 def redial(self):
 """执行断线重拨"""
 # 断开连接
 subprocess.run(['ssh', 'root@192.168.1.1', 'ifdown wan'])
 time.sleep(3)

 # 重新连接
 subprocess.run(['ssh', 'root@192.168.1.1', 'ifup wan'])
 time.sleep(5)

 # 获取新IP
 result = subprocess.run(
 ['ssh', 'root@192.168.1.1',
 "ifconfig pppoe-wan | grep 'inet addr' | cut -d: -f2 | cut -d' ' -f1"],
 capture_output=True, text=True
)
 self.current_ip = result.stdout.strip()
 return self.current_ip

 def register_ip(self):
 """向中心服务器注册新IP"""
 data = {
 'node_id': self.node_id,
 'ip': self.current_ip,
 'port': self.proxy_port,
 'location': self.location,
 'isp': self.isp
 }
 response = requests.post(f'{self.api_server}/register', json=data)
 return response.json()

 def heartbeat(self):
 """发送心跳"""
 data = {'node_id': self.node_id, 'ip': self.current_ip}
 try:
 response = requests.post(
 f'{self.api_server}/heartbeat',
 json=data,
```

```
 timeout=5
)
 return response.json()
except:
 return None

def run(self, redial_interval=300):
 """主循环"""
 while True:
 # 执行拨号获取新 IP
 new_ip = self.redial()
 print(f"New IP: {new_ip}")

 # 注册新 IP
 self.register_ip()

 # 定期发送心跳
 for _ in range(redial_interval // 10):
 time.sleep(10)
 self.heartbeat()

启动节点
if __name__ == '__main__':
 node = DialUpNode(
 api_server='http://api.example.com',
 node_id='node-shanghai-01',
 location='上海',
 isp='电信'
)
 node.run(redial_interval=300)
```

---

## 中心 API 服务器

```
from flask import Flask, request, jsonify
from datetime import datetime, timedelta
import threading
import time

app = Flask(__name__)

IP 池存储
ip_pool = {}
lock = threading.Lock()

@app.route('/register', methods=['POST'])
def register():
 """注册新IP"""
 data = request.json
 with lock:
 ip_pool[data['node_id']] = {
 'ip': data['ip'],
 'port': data['port'],
 'location': data['location'],
 'isp': data['isp'],
 'last_heartbeat': datetime.now(),
 'in_use': False
 }
 return jsonify({'status': 'ok'})

@app.route('/heartbeat', methods=['POST'])
def heartbeat():
 """接收心跳"""
 data = request.json
 with lock:
 if data['node_id'] in ip_pool:
 ip_pool[data['node_id']]['last_heartbeat'] = datetime.now()
 return jsonify({'status': 'ok'})

@app.route('/get_proxy', methods=['GET'])
def get_proxy():
 """获取可用代理"""
 location = request.args.get('location')
 isp = request.args.get('isp')

 with lock:
 for node_id, info in ip_pool.items():
 # 检查是否超时
 if datetime.now() - info['last_heartbeat'] > timedelta(seconds=30):
 continue

 # 检查是否被占用
 if info['in_use']:
 continue

 # 匹配条件
 if location and info['location'] == location:
 if isp and info['isp'] == isp:
 return jsonify({'node_id': node_id})
```

```
if location and info['location'] != location:
 continue
if isp and info['isp'] != isp:
 continue

标记为使用中
info['in_use'] = True
return jsonify({
 'proxy': f'{info["ip"]}:{info["port"]}',
 'node_id': node_id,
 'location': info['location'],
 'isp': info['isp']
})

return jsonify({'error': 'No available proxy'}), 404

@app.route('/release_proxy', methods=['POST'])
def release_proxy():
 """释放代理"""
 data = request.json
 with lock:
 if data['node_id'] in ip_pool:
 ip_pool[data['node_id']]['in_use'] = False
 return jsonify({'status': 'ok'})

if __name__ == '__main__':
 app.run(host='0.0.0.0', port=5000)
```

## 4. 业务调用示例

```
import requests

class ProxyPoolClient:
 def __init__(self, api_server):
 self.api_server = api_server
 self.current_proxy = None
 self.current_node_id = None

 def get_proxy(self, location=None, isp=None):
 """获取代理"""
 params = {}
 if location:
 params['location'] = location
 if isp:
 params['isp'] = isp

 response = requests.get(f'{self.api_server}/get_proxy', params=params)
 if response.status_code == 200:
 data = response.json()
 self.current_proxy = data['proxy']
 self.current_node_id = data['node_id']
 return self.current_proxy
 return None

 def release_proxy(self):
 """释放当前代理"""
 if self.current_node_id:
 requests.post(
 f'{self.api_server}/release_proxy',
 json={'node_id': self.current_node_id}
)
 self.current_proxy = None
 self.current_node_id = None

 def request_with_proxy(self, url, **kwargs):
 """使用代理发送请求"""
 if not self.current_proxy:
 self.get_proxy()

 proxies = {
 'http': f'http://{self.current_proxy}',
 'https': f'https://{self.current_proxy}'
 }
 return requests.get(url, proxies=proxies, **kwargs)

使用示例
client = ProxyPoolClient('http://api.example.com')

获取上海电信的代理
proxy = client.get_proxy(location='上海', isp='电信')
print(f"Using proxy: {proxy}")
```

```
发送请求
response = client.request_with_proxy('https://httpbin.org/ip')
print(response.json())

释放代理
client.release_proxy()
```

## 总结

动态住宅 IP 代理池是大规模爬虫和逆向分析的核心基础设施之一。通过合理的架构设计和管理策略，可以构建一个高可用、高伪装度的代理系统，有效绕过目标服务器的反爬虫策略。

---

## [R26] 代理池设计

# R26: 代理池设计与 Scrapy 集成

在面对反爬虫策略严厉的目标站（如电商、社交媒体）时，单一 IP 很容易被封禁。构建一个高可用、自动轮转的代理池（Proxy Pool）是大规模数据采集的基础设施。

## 1. 代理池架构设计

一个成熟的代理池系统通常包含四个独立模块，通过 Redis 进行解耦：

### 核心组件

#### 1. Fetcher (获取器):

- 职责：定时从各大免费代理网站（快代理、云代理等）或付费 API 接口拉取代理 IP。
- 策略：每隔 N 分钟运行一次，将获取到的新 IP 存入 Redis 的“待检测”队列。

#### 2. Checker (检测器):

- 职责：异步检测 Redis 中代理 IP 的可用性、匿名度和响应速度。
- 实现：使用 `aiohttp` 或 `requests` 对目标网站（如百度、谷歌或特定目标站）发起请求。
- 评分机制：

项目	说明
可用	分数设为 100 (或 +1)。
不可用	分数减 1, 当分数低于阈值 (如 0) 时, 从 Redis 移除。
复检	定时遍历 Redis 中现存的代理进行复检, 确保库中 IP 始终有效。

### 1. Storage (存储器):

项目	说明
数据库	Redis 是最佳选择。
数据结构	Sorted Set (有序集合)。
Key	代理 IP ( 1.2.3.4:8080 )
Score	代理分数 (0-100)
优势	可以利用 ZRANGEBYSCORE 轻松获取高质量 (满分) 代理。

### 1. API Server (接口服务):

- 职责: 为爬虫提供简单的 HTTP 接口获取代理。
- 接口:
  - /get : 随机返回一个高分代理。
  - /count : 查看当前可用代理数量。

## 架构图

```

graph LR
 ProxySources[免费/付费源] --> Fetcher
 Fetcher -->|Raw Proxy| Redis[(Redis Sorted Set)]
 Redis <-->|Validation| Checker
 Crawler[Scrapy 爬虫] -->|Request| API[API Server]
 API -->|Get High Score Proxy| Redis

```

## 2. Scrapy 中间件集成

### 工作流程

1. 请求前 (`process_request`): 从代理池获取一个代理, 赋值给 `request.meta['proxy']`。
2. 响应后 (`process_response`): 检查状态码。如果是 200, 说明代理正常; 如果是 403/429/超时, 说明代理可能失效或被封。
3. 异常处理 (`process_exception`): 捕获连接超时、连接拒绝等网络错误, 标记该代理失效, 并对当前请求进行重试。

---

## 代码实现

```
middlewares.py
import requests
import logging
from scrapy.exceptions import IgnoreRequest

class ProxyMiddleware:
 def __init__(self, proxy_pool_url):
 self.proxy_pool_url = proxy_pool_url
 self.logger = logging.getLogger(__name__)

 @classmethod
 def from_crawler(cls, crawler):
 return cls(
 proxy_pool_url=crawler.settings.get('PROXY_POOL_URL')
)

 def _get_random_proxy(self):
 try:
 response = requests.get(self.proxy_pool_url)
 if response.status_code == 200:
 return response.text.strip()
 except requests.ConnectionError:
 return None
 return None

 def process_request(self, request, spider):
 # 如果请求已经设置代理（例如特定请求），则跳过
 if request.meta.get('proxy'):
 return

 proxy = self._get_random_proxy()
 if proxy:
 self.logger.debug(f"Using proxy: {proxy}")
 # 设置代理，格式：http://user:pass@ip:端口 或 http://ip:端口
 request.meta['proxy'] = f"http://{proxy}"
 else:
 self.logger.warning("No proxy available from pool!")

 def process_response(self, request, response, spider):
 # 如果遇到验证码、封禁等状态码
 if response.status in [403, 429]:
 self.logger.warning(
 f"Proxy {request.meta.get('proxy')} banned "
 f"(Status {response.status}), retrying..."
)
 # 标记该代理失效（可选：调用接口报告该代理坏）
 # self._report_bad_proxy(request.meta.get('proxy'))

 # 删除当前代理设置，重新调度请求（会再次经过 process_request 换新代理）
 del request.meta['proxy']
 return request.replace(dont_filter=True)
```

```
 return response

def process_exception(self, request, exception, spider):
 # 处理连接超时、DNS 错误等
 self.logger.error(f"Proxy {request.meta.get('proxy')} failed: {exception}")

 # 换代理重试
 if 'proxy' in request.meta:
 del request.meta['proxy']
 return request.replace(dont_filter=True)
```

## 配置 settings.py

```
settings.py
DOWNLOADER_MIDDLEWARES = {
 'myproject.middlewares.ProxyMiddleware': 543,
 # 禁用 Scrapy 默认 UserAgent 和重试中间件（视情况而定）
 # 'scrapy.downloadermiddlewares.userAgent.UserAgentMiddleware': None,
}

PROXY_POOL_URL = 'http://localhost:5000/get'
```

## 3. 开源代理池推荐

### 1. proxy\_pool

- GitHub: [jhao104/proxy\\_pool](#)
- 特点: 也是基于 Redis, 架构清晰, 支持 Docker 一键部署, 内置了几十个免费源的抓取规则。

### 2. Gerapy / Scylla

- GitHub: [imWildCat/scylla](#)
- 特点: 智能化代理池, 自动学习代理的稳定性。

### 3. GimmeProxy

- 特点: Go 语言编写, 性能强劲。

## 4. 隧道代理 (Tunnel Proxy)

对于企业级应用，维护自建代理池成本较高（免费 IP 质量极差，可用率不足 5%）。此时通常使用厂商提供的隧道代理。

特点：

- 不需要在本地维护 IP 池。
- 只有一个固定的入口地址（如 `http://proxy.vendor.com:8000`）。
- 每一次请求，云端会自动转发给背后不同的动态 IP。

Scrapy 集成：

只需要在 `process_request` 中将代理设置为该固定地址，并在 Header 中添加鉴权信息。

```
Tunnel Proxy Example
import base64

def process_request(self, request, spider):
 request.meta['proxy'] = "http://proxy.vendor.com:8000"
 # 某些厂商要求在头部通过 Proxy-Authorization 认证
 auth = base64.b64encode(b"user:pass").decode()
 request.headers['Proxy-Authorization'] = f"Basic {auth}"
```

## 总结

代理池是大规模爬虫系统的核心基础设施。通过合理的架构设计（Fetcher → Checker → Storage → API）和 Scrapy 中间件集成，可以构建一个高可用、自动轮转的代理系统，有效应对目标站的反爬虫策略。

---

## [R27] Scrapy 爬虫框架

# R27: Scrapy 快速入门备忘录

Scrapy 是一个用于网络爬虫和数据抓取的、开源的、协作式的 Python 框架。它具有速度快、功能强大、可扩展性高的特点。本备忘录为 Scrapy 的核心概念和常用命令提供快速参考。

## 目录

- Scrapy 快速入门备忘录
  - 目录
    - 核心组件
    - 项目命令
    - Spider (爬虫)
      - 基本结构
      - 处理分页和链接

## 核心组件

Scrapy 的数据流由以下核心组件协同完成：

1. Engine (引擎): 负责控制所有组件之间的数据流，并在相应动作发生时触发事件。
2. Scheduler (调度器): 接收来自引擎的请求 (Request)，并将其入队，以便后续引擎请求时提供。
3. Downloader (下载器): 负责获取页面数据，并将其提供给引擎，而后由引擎将结果 (Response) 交给 Spider。
4. Spiders (爬虫): 用户编写的用于解析 Response 并提取 Item 或额外 Request 的类。

5. Item Pipeline (项目管道): 负责处理由 Spider 提取出来的 `Item`。典型的操作包括数据清洗、验证和持久化（如存入数据库）。
6. Downloader Middlewares (下载器中间件): 位于引擎和下载器之间的钩子，用于在请求发送和响应返回时进行自定义处理（如设置 User-Agent、处理代理）。
7. Spider Middlewares (爬虫中间件): 位于引擎和 Spider 之间的钩子，用于处理 Spider 的输入 (`Response`) 和输出 (`Item`, `Request`)。

## Scrapy Architecture

### 项目命令

命令	描述
<code>pip install scrapy</code>	安装 Scrapy 框架
<code>scrapy startproject myproject</code>	创建一个名为 <code>myproject</code> 的新项目
<code>cd myproject</code>	进入项目目录
<code>scrapy genspider example example.com</code>	在 <code>spiders</code> 目录下创建一个名为 <code>example</code> 的爬虫，限定域名为 <code>example.com</code>
<code>scrapy crawl example</code>	运行名为 <code>example</code> 的爬虫
<code>scrapy crawl example -o output.json</code>	运行爬虫并将提取的数据保存为 JSON 文件
<code>scrapy shell "http://example.com"</code>	启动一个交互式 Shell，用于测试 XPath/CSS 选择器
<code>scrapy list</code>	列出项目中的所有可用爬虫

## Spider (爬虫)

Spider 是你定义如何爬取某个网站（或一组网站）的类，包括爬取动作和如何从页面内容中提取结构化数据。

### 基本结构

```
myproject/spiders/example_spider.py
import scrapy

class ExampleSpider(scrapy.Spider):
 # 爬虫的唯一标识名称
 name = 'example'
 # 允许爬取的域名列表（可选）
 allowed_domains = ['example.com']
 # 爬虫启动时请求的 URL 列表
 start_urls = ['http://example.com/']

 # 处理 start_urls 响应的默认回调方法
 def parse(self, response):
 # 在这里编写解析逻辑
 pass
```

- `response.css('a::attr(href)').getall()`：提取所有 `<a>` 标签的 `href` 属性。
- `response.css('div.product > p::text').get()`：提取 `class="product"` 的 `div` 下的 `p` 标签文本。
- XPath 表达式：
  - `response.xpath('//h1/text()').get()`：提取第一个 `<h1>` 标签的文本。
  - `response.xpath('//a/@href').getall()`：提取所有 `<a>` 标签的 `href` 属性。
  - `response.xpath('//div[@class="product"]/p/text()').get()`：同上。

### 处理分页和链接

在 `parse` 方法中，你可以 `yield` 新的 `Request` 对象来跟进链接。

```
def parse(self, response):
... 提取当前页面数据 ...

提取下一页链接并生成新请求
next_page = response.css('a.next_page::attr(href)').get()
if next_page is not None:
response.urljoin() 用于处理相对URL
yield response.follow(next_page, callback=self.parse)
```

```
myproject/items.py
import scrapy

class ProductItem(scrapy.Item):
name = scrapy.Field()
price = scrapy.Field()
description = scrapy.Field()
```

```
item = ProductItem() item['name'] = response.css('h1.product-name::text').get()
item['price'] = response.css('span.price::text').get() yield item
```

```
```python
# myproject/pipelines.py
import sqlite3

class SQLitePipeline:
def open_spider(self, spider):
# 爬虫开启时调用
self.connection = sqlite3.connect('products.db')
self.cursor = self.connection.cursor()
self.cursor.execute('CREATE TABLE IF NOT EXISTS products (name TEXT, price TEXT)')

def close_spider(self, spider):
# 爬虫关闭时调用
self.connection.close()

def process_item(self, item, spider):
# 每个 item 都会调用
self.cursor.execute('INSERT INTO products (name, price) VALUES (?, ?)', (item['name'],
item['price']))
self.connection.commit()
return item # 必须返回 item
```

```
'myproject.pipelines.SQLitePipeline': 300, }
```

- * `DEFAULT_REQUEST_HEADERS`： 设置默认的请求头，如 `User-Agent`。
- * `DOWNLOAD_DELAY = 1`： 设置下载延迟（秒），以避免对服务器造成太大压力。
- * `CONCURRENT_REQUESTS = 16`： 并发请求数。
- * `ITEM_PIPELINES`： 激活和设置 Item Pipeline 的优先级。
- * `DOWNLOADER_MIDDLEWARES`： 激活和设置下载器中间件的优先级。

[R28] Scrapy-Redis 分布式

R28: 分布式爬虫实战: Scrapy-Redis 详解

Scrapy 默认是单机架构, 请求队列保存在内存中, 重启即失, 且无法多机共享。Scrapy-Redis 是一个强大的组件, 它重写了 Scrapy 的调度器 (Scheduler) 和去重组件 (DupeFilter), 将请求队列和指纹集合存储在 Redis 中, 从而实现:

1. 分布式爬取: 多个爬虫节点共享同一个 Redis 队列, 协同工作。
2. 断点续爬: 请求持久化在 Redis 中, 爬虫挂掉重启后可继续运行。

1. 核心架构原理

原生 Scrapy vs Scrapy-Redis

原生 Scrapy:

- Scheduler: 维护在内存中的 Python `deque` 或 `queue`。
- DupeFilter: 维护在内存中的 Python `set`。
- 缺点: 无法跨进程/跨机器共享, 内存受限。

Scrapy-Redis:

- Scheduler: 从 Redis 的 `List` (或 `PriorityQueue`) 中 `POP` 请求, 向其 `PUSH` 新请求。
- DupeFilter: 利用 Redis 的 `Set` 数据结构存储 URL 指纹 (SHA1), 利用 Redis 的原子性进行去重。
- Item Pipeline: 可选将提取的数据直接推入 Redis, 由独立的 Worker 消费存储。

2. 环境搭建与配置

安装

```
pip install scrapy-redis
```

配置 settings.py

```
# settings.py

# 1. 启用 Scrapy-Redis 调度器
SCHEDULER = "scrapy_redis.scheduler.Scheduler"

# 2. 启用 Scrapy-Redis 去重过滤器
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"

# 3. 允许暂停 (断点续爬核心)
# 如果为真, 当爬虫停止时, Redis 中的请求队列不会被清空
SCHEDULER_PERSIST = True

# 4. 设置 Redis 连接
# 方式一: 单独设置
REDIS_HOST = '192.168.1.100'
REDIS_PORT = 6379
# REDIS_PARAMS = {'password': 'yourpassword'}

# 方式二: 完整地址
# REDIS_URL = 'redis://user:pass@hostname:9001'

# 5. 配置请求队列模式 (可选, 默认为 PriorityQueue)
# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.PriorityQueue' # 有序集合, 支持优先级
# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.FifoQueue' # 先进先出列表
# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.LifoQueue' # 后进先出列表 (栈)

# 6. (可选) 将数据存入 Redis Pipeline
ITEM_PIPELINES = {
    'scrapy_redis.pipelines.RedisPipeline': 300,
}
```

3. 编写分布式 Spider

继承 RedisSpider

```
from scrapy_redis.spiders import RedisSpider
import scrapy

class MyDistributedSpider(RedisSpider):
    name = 'myspider_distributed'

    # 核心差异: 不再定义 start_urls
    # 定义 redis_key, 爬虫启动后会阻塞等待该键中出现的 URL
    redis_key = 'myspider:start_urls'

    def parse(self, response):
        self.logger.info(f"Crawling {response.url}")

        # 提取数据逻辑与普通 Spider 一致
        yield {
            'url': response.url,
            'title': response.css('title::text').get()
        }

        # 生成新请求
        for href in response.css('a::attr(href)').getall():
            yield response.follow(href, self.parse)
```

启动流程

1. 启动爬虫（多个终端启动多个实例）：

```
scrapy crawl myspider_distributed
```

1. 向 Redis 推送起始 URL:

```
redis-cli lpush myspider:start_urls http://example.com
```

使用 RedisCrawlSpider

如果你需要利用 `Rule` 和 `LinkExtractor` 自动抓取全站，可以使用 `RedisCrawlSpider`。

```
from scrapy_redis.spiders import RedisCrawlSpider
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import Rule

class MyCrawlSpider(RedisCrawlSpider):
    name = 'mycrawl_distributed'
    redis_key = 'mycrawl:start_urls'

    rules = (
        Rule(LinkExtractor(allow=r'/article/'), callback='parse_article'),
        Rule(LinkExtractor(allow=r'/page/'), follow=True),
    )

    def parse_article(self, response):
        yield {
            'title': response.css('h1::text').get(),
            'content': response.css('.content::text').getall(),
        }
```

4. 进阶优化策略

Bloom Filter 去重优化

Scrapy-Redis 默认使用 Redis `Set` 存储所有指纹。对于亿级 URL 的爬取，这会消耗数十 GB 内存。解决方案是集成 Bloom Filter。

实现思路：

1. 重写 `RFPDupeFilter`。
2. 使用 `redis-py` 的 `bf.add` 和 `bf.exists` 命令（需要 `RedisBloom` 模块）或 Python 端的 `pybloom_live` 映射到 Redis BitMap。

```
# custom_dupefilter.py 简易示意
from scrapy_redis.dupefilter import RFPDupeFilter

class BloomFilterDupeFilter(RFPDupeFilter):
    def request_seen(self, request):
        fp = self.request_fingerprint(request)
        # 假设 self.server 是 Redis 连接, 且支持 BF 命令
        # 如果指纹已存在, 返回 True
        if self.server.execute_command('BF.EXISTS', self.key, fp):
            return True
        # 否则添加指纹
        self.server.execute_command('BF.ADD', self.key, fp)
        return False
```

请求优先级

```
# 生成请求时指定优先级
yield scrapy.Request(url, priority=100)  # 首页, 优先
yield scrapy.Request(url, priority=10)   # 详情页, 次之
```

空闲超时关闭

```
# settings.py
# 指定空闲等待时间(秒), 超时则关闭爬虫
SCHEDULER_IDLE_BEFORE_CLOSE = 10
```

5. 部署架构图

```
graph TB
    subgraph "Master / Redis Server"
        Redis["Redis Queue & Set"]
    end

    subgraph "Slave 1"
        Spider1["Scrapy Spider 1"]
        Spider1 -->|Pop Request| Redis
        Spider1 -->|Push Request| Redis
        Spider1 -->|Dupe Check| Redis
    end

    subgraph "Slave 2"
        Spider2["Scrapy Spider 2"]
        Spider2 -->|Pop Request| Redis
        Spider2 -->|Push Request| Redis
        Spider2 -->|Dupe Check| Redis
    end

    subgraph "Data Storage"
        Mongo["MongoDB"]
    end

    Spider1 -->|Store Item| Mongo
    Spider2 -->|Store Item| Mongo
```

6. 常见问题

Q: 爬虫启动后一直等待，不抓取？

A: 确认已向 `redis_key` 推送了起始 URL:

```
redis-cli lpush myspider:start_urls http://example.com
```

Q: 如何监控爬取进度？

A: 使用 Redis 命令查看队列长度:

```
# 查看待爬取请求数量  
redis-cli llen myspider:requests  
  
# 查看已爬取指纹数量  
redis-cli scard myspider:dupefilter
```

Q: 如何清空队列重新开始?

A: 删除 Redis 中的相关键:

```
redis-cli del myspider:requests myspider:dupefilter myspider:start_urls
```

总结

Scrapy-Redis 是构建分布式爬虫系统的核心组件。通过将调度器和去重组件迁移到 Redis，实现了多节点协同爬取和断点续爬能力，是大规模数据采集的必备工具。

[R29] Docker 部署

R29: 容器化部署: Docker 与 Kubernetes 实战

将爬虫项目容器化是实现标准化部署、弹性伸缩和 CI/CD 的第一步。本指南将详细介绍如何编写 Dockerfile，使用 Docker Compose 编排服务，以及如何在 Kubernetes (K8s) 上运行爬虫任务。

1. Dockerfile 最佳实践

我们需要为 Scrapy 项目构建一个轻量、稳定的 Docker 镜像。

目录结构

```
my_crawler/
├── scrapy.cfg
├── requirements.txt
└── Dockerfile
└── myproject/
    ├── __init__.py
    ├── items.py
    ├── settings.py
    └── spiders/
```

Dockerfile

```
FROM python:3.9-slim-buster

# 设置工作目录
WORKDIR /app

# 设置环境变量
# 防止 Python 生成 .pyc 文件
ENV PYTHONDONTWRITEBYTECODE 1
# 防止 Python 缓冲区 stdout/stderr, 确保日志实时输出
ENV PYTHONUNBUFFERED 1
# 设置时区 (可选)
ENV TZ=Asia/Shanghai

# 安装系统依赖 (如果需要编译 lxml 或其它库)
# RUN apt-get update && apt-get install -y gcc libxml2-dev libxslt-dev && rm -rf /var/lib/apt/lists/*

# 复制依赖文件并安装
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 复制项目代码
COPY ..

# 默认启动命令 (可被 docker run 覆盖)
# 这里我们默认启动 scrapyd (如果使用 scrapyd 部署) 或仅作为一个 shell
CMD ["scrapy", "list"]
```

构建与运行

```
# 构建镜像
docker build -t my-crawler:v1 .

# 运行爬虫
docker run --rm my-crawler:v1 scrapy crawl myspider
```

2. Docker Compose 编排

对于分布式 Scrapy-Redis 架构，我们需要同时运行 Redis、MongoDB 和多个爬虫节点。

docker-compose.yml

```
version: '3.8'

services:
    # 1. Redis 服务 (消息队列)
    redis:
        image: redis:6.2-alpine
        ports:
            - "6379:6379"
        volumes:
            - redis_data:/data
        command: redis-server --appendonly yes

    # 2. MongoDB 服务 (数据存储)
    mongo:
        image: mongo:5.0
        ports:
            - "27017:27017"
        environment:
            MONGO_INITDB_ROOT_USERNAME: admin
            MONGO_INITDB_ROOT_PASSWORD: password
        volumes:
            - mongo_data:/data/db

    # 3. 爬虫服务 (Master/Slave 模式中的 Slave)
    crawler:
        build: .
        image: my-crawler:latest
        # 覆盖默认命令, 启动爬虫
        command: scrapy crawl myspider_distributed
        # 依赖服务就绪
        depends_on:
            - redis
            - mongo
        environment:
            - REDIS_HOST=redis
            - MONGO_URI=mongodb://admin:password@mongo:27017
        # 想要开启多个爬虫节点? 直接 scale
        deploy:
            replicas: 3

volumes:
    redis_data:
    mongo_data:
```

启动与扩容

```
# 启动所有服务
docker-compose up -d

# 扩容爬虫节点到 5
docker-compose up -d --scale crawler=5

# 查看日志
docker-compose logs -f crawler

# 停止所有服务
docker-compose down
```

3. Kubernetes 部署

在 K8s 上运行爬虫，可以利用其强大的调度和自愈能力。

Deployment (持续运行的爬虫 Worker)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: scrapy-worker
  labels:
    app: scrapy-worker
spec:
  # 副本数, 即并发爬虫节点数量
  replicas: 5
  selector:
    matchLabels:
      app: scrapy-worker
  template:
    metadata:
      labels:
        app: scrapy-worker
    spec:
      containers:
        - name: crawler
          image: registry.example.com/my-crawler:v1
          # 容器启动命令
          command: ["scrapy", "crawl", "myspider_distributed"]
          # 环境变量配置
          env:
            - name: REDIS_HOST
              value: "redis-service" # K8s Service Name
            - name: MONGO_URI
              valueFrom:
                secretKeyRef:
                  name: db-secrets
                  key: mongo-uri
      resources:
        requests:
          memory: "256Mi"
          cpu: "250m"
        limits:
          memory: "512Mi"
          cpu: "500m"
```

CronJob (定时任务)

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-crawler
spec:
  # 每天凌晨 2 点运行
  schedule: "0 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: crawler
              image: registry.example.com/my-crawler:v1
              command: ["scrapy", "crawl", "daily_spider"]
              env:
                - name: REDIS_HOST
                  value: "redis-service"
  restartPolicy: OnFailure
```

常用 kubectl 命令

```
# 应用配置
kubectl apply -f crawler-deployment.yaml

# 查看 Pod 状态
kubectl get pods

# 动态扩缩容（无需修改 yaml）
kubectl scale deployment scrapy-worker --replicas=10

# 查看日志
kubectl logs -f deployment/scrapy-worker

# 进入容器调试
kubectl exec -it <pod-name> -- /bin/bash
```

4. 集成 Scrapyd 管理平台

Scrapyd: Scrapy 官方的部署服务，提供 HTTP API 来部署、启动、停止爬虫。

Gerapy: 基于 Scrapyd 的分布式管理 GUI, 支持节点管理、代码编辑、定时任务。

Dockerfile (集成 Scrapyd)

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install scrapyd scrapy-redis
COPY scrapyd.conf /etc/scrapyd/
EXPOSE 6800
CMD ["scrapyd"]
```

scrapyd.conf

```
[scrapyd]
eggs_dir      = eggs
logs_dir       = logs
items_dir      = items
jobs_to_keep  = 5
dbs_dir        = dbs
max_proc       = 0
max_proc_per_cpu = 4
finished_to_keep = 100
poll_interval = 5.0
bind_address  = 0.0.0.0
http_port     = 6800
debug         = off
runner        = scrapyd.runner
application   = scrapyd.app.application
launcher      = scrapyd.launcher.Launcher
webroot       = scrapyd.website.Root
```

部署爬虫到 Scrapyd

```
# 打包项目
scrapyd-deploy -p myproject

# 启动爬虫
curl http://localhost:6800/schedule.json -d project=myproject -d spider=myspider

# 查看爬虫状态
curl http://localhost:6800/listjobs.json?project=myproject
```

5. CI/CD 集成

GitHub Actions 示例

```
name: Build and Deploy Crawler

on:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Build Docker image
        run: docker build -t my-crawler:${{ github.sha }} .

      - name: Push to Registry
        run: |
          docker login -u ${{ secrets.DOCKER_USER }} -p ${{ secrets.DOCKER_PASS }}
          docker tag my-crawler:${{ github.sha }} registry.example.com/my-crawler:${
            {{ github.sha }}}
          docker push registry.example.com/my-crawler:${{ github.sha }}

      - name: Deploy to K8s
        run: |
          kubectl set image deployment/scrapy-worker crawler=registry.example.com/my-
crawler:${{ github.sha }}
```

总结

容器化部署是现代爬虫工程的标准实践。通过 Docker 实现环境一致性，通过 Docker Compose 简化本地开发，通过 Kubernetes 实现生产环境的弹性伸缩和自愈能力。结合 Scrapyd 和 CI/CD 流程，可以构建一个完整的爬虫 DevOps 体系。

[R30] 虚拟化与容器

R30: 工程化：虚拟化与容器技术

移动端虚拟化技术是在服务器端模拟出成百上千个 Android 设备环境的能力，它是所有大规模自动化测试、数据采集和安全分析任务的基石。这项技术的核心在于平衡性能、隔离性和真实性。

1. Android 模拟器 (Emulators)

Android 模拟器是在非 ARM 架构（通常是 x86_64）的服务器上，通过软件来模拟一个完整的 Android 操作系统环境的程序。

a) 主流方案对比

方案	类型	优点	缺点	适用场景
Android SDK Emulator	全系统模拟 (QEMU)	功能最全, Google 官方支持, 能模拟最新的 Android API 和 Google 服务。	性能开销极大, 资源占用高, 启动慢。	应用开发与调试, 小规模测试。
Anbox / Waydroid	基于内核的容器	性能极高, 接近原生, 资源占用小。	隔离性较差, 依赖宿主机内核, 可能存在兼容性问题。	对性能要求极高的游戏测试, 云游戏。
Redroid / anbox-cloud	基于 Docker 的容器化模拟器	部署、扩展和管理极为方便, 易于集成到 CI/CD 流水线。	配置相对复杂, 对 Docker 和网络知识有一定要求。	大规模、动态伸缩的云手机平台, 自动化测试集群。
Genymotion	商业虚拟化方案	性能好, 功能强大 (GPS, 摄像头模拟), 提供 PaaS/SaaS 服务。	收费, 闭源。	企业级测试, 需要专业技术支持的场景。

b) 关键技术点

- 指令集翻译: 在 x86 服务器上运行为 ARM 架构编译的 App, 需要进行动态的指令集翻译。Intel 的 `libhoudini` 和 Google 的 `NdkVm` 是实现这一功能的关键组件, 其效率直接决定了模拟器的性能。
- GPU 硬件加速: 为了渲染复杂的 UI 和游戏, 模拟器需要将 Android 的图形渲染指令 (OpenGL ES) 桥接到宿主机的 GPU 上。`Virgil3D` 等项目实现了这种透传能力。
- 快照与状态管理: 高效的快照功能允许我们快速地将模拟器恢复到一个干净、预设的状态, 这对于保证每次自动化测试都在同样的环境中进行至关重要。

2. 容器技术 (Containerization)

容器技术（以 Docker 为代表）虽然不直接运行 Android 系统，但它在整个工程化体系中扮演着“胶水”和“标准交付”的关键角色。

a) 隔离与打包依赖项

在复杂的自动化流程中，除了 Android 模拟器本身，我们还需要大量的周边服务。

- 任务队列: 使用 `redis` 或 `rabbitmq` 容器来管理和分发成千上万的测试任务。
- 代理服务: 部署 `mitmproxy` 或 `squid` 容器来集中拦截和分析所有模拟器的网络流量。
- 数据库: 使用 `mongodb` 或 `postgresql` 容器来持久化存储测试结果、App 元数据和设备状态。
- 文件存储: 使用 `minio` 容器来提供一个 S3 兼容的对象存储服务，用于存放 APK 文件、测试报告和截图。

将这些服务全部容器化，意味着我们可以通过一个 `docker-compose.yml` 文件，在任何环境中一键拉起整套后端基础设施，极大地简化了部署和运维。

b) 构建标准化的执行环境

我们可以将 Appium、UIAutomator2 脚本、Frida 脚本以及所有 Python 依赖打包到一个 Docker 镜像中。

- 一致性: 确保无论是在开发者的本地机器上，还是在 CI/CD 服务器上，脚本的运行环境都完全一致，避免了“在我这里能跑”的问题。
- 版本控制: 可以为每个版本的 App 配套一个特定版本的测试镜像，方便地对历史版本进行回溯测试。
- 可移植性: 整个测试套件可以作为一个 Docker 镜像轻松地迁移到不同的云平台或物理服务器上。

总结

虚拟化和容器化是从"手工作坊"迈向"工业化生产"的第一步。

- 虚拟化 解决了"设备从哪里来"的问题，提供了可大规模复制的、隔离的 Android 运行环境。
- 容器化 解决了"依赖和脚本如何管理"的问题，提供了标准化的、可移植的交付物。

二者结合，为上层的自动化和群控系统提供了坚实、可靠、可扩展的基础设施。

[R31] Web 反爬技术

R31: Web 反爬虫技术

网络爬虫是自动从网站提取数据的过程。由于这可能被滥用，许多现代网站和服务实施了复杂的反爬虫（或“反机器人”）技术来保护其数据。这些技术可大致分为客户端（浏览器）和服务器端防御。

1. 客户端（浏览器）挑战

这些防御措施的重点是确保客户端是由人类操作的真实、标准的网络浏览器。

- JavaScript (JS) 挑战: 服务器发送一段复杂的 JavaScript，客户端必须正确执行。该脚本可能执行计算、与浏览器 API 交互，并生成一个必须在后续请求中发回的令牌。需要使用像 Puppeteer 或 Selenium 这样的无头浏览器来通过这些挑战。
 - 例子: Akamai Bot Manager, Cloudflare Bot Management。
- 浏览器指纹: 服务器收集客户端浏览器环境的详细画像。这包括：
 - User-Agent、屏幕分辨率、颜色深度、时区。
 - 安装的字体、浏览器插件。
 - 浏览器 JS 引擎或渲染引擎 (Canvas) 行为的细微差异。
- TLS 指纹: 分析客户端 TLS 握手的参数（密码套件、扩展等）。
 - 与标准浏览器画像（例如，Windows 上的 Chrome）的偏差可用于将客户端标记为机器人。

- CAPTCHA: "全自动区分计算机和人类的公开图灵测试"。这需要用户解决一个对机器人来说很困难的挑战（例如，在图像中识别物体）。
 - 例子: 谷歌的 reCAPTCHA (v2/v3), hCaptcha。绕过这些通常需要使用第三方破解服务。

2. 服务器端检测

这些防御措施通过分析服务器上的请求模式来识别非人类行为。

- IP 地址信誉: 阻止或速率限制来自已知属于数据中心（如 AWS、谷歌云）或代理/VPN 服务的 IP 地址的请求。通常使用住宅代理来规避此问题。
- 速率限制: 限制单个 IP 地址或用户帐户在给定时间段内可以发出的请求数量。爬虫必须遵守这些限制以避免被阻止。
- 行为分析: 这是最先进的技术。服务器会长期跟踪用户行为，以建立"正常"人类交互的模型。
 - 鼠标移动和按键: 真实用户有混乱、非线性的鼠标移动和打字模式。机器人通常缺乏这一点。高级机器人必须模拟这种"人类"输入。
 - 导航模式: 人类通过网站的路径是可预测但非完全线性的。机器人通常直接或以僵硬的顺序访问页面。
 - 请求时间: 来自人类用户的请求之间的时间是可变的。机器人通常以固定的、最小的延迟运行。

规避策略

- 使用功能齐全的浏览器: 使用 Selenium、Puppeteer 或 Playwright 等工具来自动化一个真实的浏览器。这有助于解决 JS 挑战并提供更具说服力的浏览器指纹。使用"stealth"插件进一步隐藏自动化。
- 轮换 IP: 使用高质量的住宅或移动代理池来避免基于 IP 的封锁并模仿真实用户。
- 模仿人类行为: 引入随机延迟，模拟逼真的鼠标移动，并以更像人类的方式在网站上导航。

- 逆向工程 JS: 对于某些 JS 挑战, 可以逆向工程混淆的 JavaScript 代码, 以了解反机器人令牌是如何生成的。这使你可以在自己的脚本中复制逻辑, 而无需完整的浏览器, 这样会快得多。甚至可以使用 Frida 等工具来钩住浏览器进程以进行分析。
- IP 质量检测: 服务器端会检查请求 IP 的类型 (数据中心、住宅、移动), 并对来自数据中心的 IP 施加更严格的限制。
- 行为分析: 服务器通过分析用户在一系列请求中的行为模式 (如请求频率、访问路径、鼠标移动轨迹) 来判断其是否为机器人。

专题：绕过 Cloudflare 五秒盾

Cloudflare 的"I'm Under Attack Mode" (我正遭受攻击模式) 是一个非常常见的强力反机器人措施, 用户会看到一个持续约五秒的"Checking your browser before accessing..."页面。这就是俗称的"五秒盾"。

1. 工作原理

五秒盾的核心是一个 JavaScript 挑战 (JS Challenge)。当用户首次访问受保护的页面时, 服务器会返回一个包含复杂、高度混淆的 JavaScript 代码的 HTML 页面。这段 JS 的主要目的不是为了好看, 而是为了:

1. 环境检测: 检查当前环境是否为一个真实的、标准的浏览器。它会检测 `window`, `document` 等对象, 以及屏幕分辨率、时区、插件等浏览器指纹信息。
2. 计算密集型任务: 执行一系列复杂的数学运算。这些运算对于现代浏览器来说耗时很短 (通常在 1-2 秒内), 但对于不具备 JS 执行引擎的简单爬虫 (如纯粹的 `requests` 库) 来说是无法完成的。
3. 生成验证 Token: JS 计算的最终结果会作为一个 Token, 通过表单提交或 Ajax 请求发送回 Cloudflare 的服务器进行验证。
4. 设置身份 Cookie: 验证通过后, Cloudflare 会在用户的浏览器中设置一个特殊的 Cookie (如 `_cf_bm` 或 `cf_clearance`) , 该 Cookie 在一定时间内有效。后续的请求只要携带这个有效的 Cookie, 就可以直接访问网站, 无需再次挑战。

2. 绕过方案

绕过五秒盾的核心思想是 模拟一个能够成功执行其 JS 挑战的环境。

a. 方案一：使用无头浏览器 (Headless Browser) - 推荐

这是最稳定、成功率最高的方案。使用 `Puppeteer` (Node.js), `Playwright` (Python/Node.js) 或 `Selenium` 等自动化浏览器框架。

- 工作方式: 这些工具会启动一个真实的、完整的浏览器内核 (如 Chrome)，只是没有图形界面。当它们访问目标页面时，浏览器会像正常用户访问一样，自动执行所有的 JavaScript，完成挑战，获取 Cookie，然后继续访问目标页面。
- 优点: 成功率极高，几乎能应对所有基于 JS 挑战的防护。
- 缺点: 资源消耗大 (需要启动整个浏览器)，速度相对较慢。

Playwright (Python) 示例：

脚本集合

[R32] Frida 脚本示例

R32: Frida 实战脚本集

本文档收集了适用于各种常见场景的 Frida 脚本。这些脚本旨在作为即用型模板，您可以根据特定目标进行修改。

目录

1. 信息收集 (Information Gathering)
 2. Hook 与修改 (Hooking & Modification)
 3. 网络监控与绕过 (Networking)
 4. 数据持久化与脱壳 (Storage & Dumping)
 5. 反调试与环境检测绕过 (Anti-Analysis)
 6. UI 与事件 (UI & Events)
-

信息收集

1. 枚举指定类的所有方法

```
// Usage: frida -U -f com.example.app -l list_methods.js
// Replace 'com.example.target class' below
Java.perform(function () {
    var targetClass = "com.example.TargetClass";
    var wrapper = Java.use(targetClass);
    var ownMethods = wrapper.class.getDeclaredMethods();

    console.log("Methods of class " + targetClass + ":");
    ownMethods.forEach(function (method) {
        console.log(method.toString());
    });
});
```

```
console.log("Listing all loaded classes..."); Java.enumerateLoadedClasses({ onMatch:
function(className) { console.log(className); }, onComplete: function()
{ console.log("Class enumeration complete."); } });});
```

```
Java.perform(function() {
var TargetClass = Java.use('com.example.app.CryptoUtils');
var methodName = 'encrypt'; // Method name to trace

// Handle method overloads
TargetClass[methodName].overloads.forEach(function(overload) {
overload.implementation = function() {
console.log('\n[+] Called ' + TargetClass.$className + '.' + methodName);

// Print arguments
for (var i = 0; i < arguments.length; i++) {
console.log(' - Argument ' + i + ': ' + arguments[i]);
}

// Call original method
var retval = this[methodName].apply(this, arguments);

// Print return value
console.log(' - Return value: ' + retval);

return retval;
};
});
```

```
Java.perform(function() {  
    var PremiumUtils = Java.use('com.example.app.PremiumUtils');  
  
    PremiumUtils.isUserPremium.implementation = function() { console.log('[+] Bypassing Premium check...'); return true; // Return true directly };});
```

```
Java.choose('com.example.app.UserInfo', {  
    onMatch: function(instance) {  
        console.log('[+] Found UserInfo instance.');// Directly modify field value  
        instance.userLevel.value = 99;  
        console.log(' - Patched userLevel to 99.');//  
    },  
    onComplete: function() {}  
});  
});
```

```
// Usage: frida -U --no-pause -f com.example.app -l universal_ssl_unpinning.js
// Source: https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-
// bypass-with-frida/
setTimeout(function () {
    Java.perform(function () {
        console.log("");
        console.log("[.] Android SSL Pinning Bypass");

        var CertificateFactory = Java.use("java.security.cert.CertificateFactory");
        var FileInputStream = Java.use("java.io.FileInputStream");
        var BufferedInputStream = Java.use("java.io.BufferedInputStream");
        var X509Certificate = Java.use("java.security.cert.X509Certificate");
        var KeyStore = Java.use("java.security.KeyStore");
        var TrustManagerFactory = Java.use("javax.net.ssl.TrustManagerFactory");
        var SSLContext = Java.use("javax.net.ssl.SSLContext");

        // TrustManagerImpl (android > 7)
        try {
            var TrustManagerImpl = Java.use(
                "com.android.org.conscrypt.TrustManagerImpl"
            );
            TrustManagerImpl.verifyChain.implementation = function (
                untrustedChain,
                trustAnchorChain,
                host,
                clientAuth,
                ocspData,
                tlsSctData
            ) {
                console.log(
                    "[+] Bypassing TrustManagerImpl verifyChain() for host: " + host
                );
                return untrustedChain;
            };
        } catch (e) {
            console.log("[-] TrustManagerImpl not found. Skipping.");
        }

        // OkHttp3
        try {
            var OkHttpClient = Java.use("okhttp3.OkHttpClient");
            OkHttpClient.Builder.prototype.build.implementation = function () {
                var builder = this.build.call(this);
                console.log("[+] OkHttp3 CertificatePinner removed.");
                builder.certificatePinner.value = null;
                return builder;
            };
        } catch (e) {
            console.log("[-] OkHttp3 not found. Skipping.");
        }

        // TrustManager (universal)
```

```

var TrustManager = Java.use("javax.net.ssl.X509TrustManager");
var checkServerTrusted = TrustManager.checkServerTrusted;
checkServerTrusted.overload(
    "java.security.cert.X509Certificate[]",
    "java.lang.String"
).implementation = function (chain, authType) {
    console.log(
        "[+] Bypassing TrustManager checkServerTrusted for authType: " +
        authType
    );
    return;
};

},
),
{
}, 0);

```

```

Java.perform(function() {
    var sharedPrefsEditor =
        Java.use('android.app.SharedPreferencesImpl$EditorImpl');

    sharedPrefsEditor.putString.implementation = function(key, value) { console.log('[SP Write] key: ' + key + ', value: ' + value); return this.putString(key, value); };

    var sharedPrefs =
        Java.use('android.app.SharedPreferencesImpl');
    sharedPrefs.getString.implementation = function(key, defValue) { var value =
        this.getString(key, defValue); console.log('[SP Read] key: ' + key + ', value: ' + value); return value; }; });

```

```

SQLiteDatabase.query.overload('java.lang.String', '[Ljava.lang.String;',
    'java.lang.String', '[Ljava.lang.String;', 'java.lang.String', 'java.lang.String',
    'java.lang.String').implementation = function(table, columns, selection,
selectionArgs, groupBy, having, orderBy) {
    console.log("\n[SQL Query] Table: " + table);
    console.log(" - Columns: " + columns);
    console.log(" - Selection: " + selection);
    console.log(" - Selection Args: " + selectionArgs);
    return this.query(table, columns, selection, selectionArgs, groupBy, having, orderBy);
};

SQLiteDatabase.execSQL.overload('java.lang.String',
    '[Ljava.lang.Object;').implementation = function(sql, bindArgs) {
    console.log("\n[SQL execSQL] SQL: " + sql);
    console.log(" - Bind Args: " + bindArgs);
    return this.execSQL(sql, bindArgs);
};
});

```

```
Java.perform(function () { var RootCheckClass =  
Java.use('com.example.security.RootUtil'); // Replace with target class  
  
RootCheckClass.isDeviceRooted.implementation = function() { console.log('[+]  
Bypassing root check...'); return false; }; });
```

```
Debug.isDebugEnabledConnected.implementation = function() {  
console.log('[+] Bypassing isDebugEnabledConnected check...');  
return false;  
}  
});
```

```
Java.perform(function() { var View = Java.use('android.view.View');  
  
View.setOnClickListener.implementation = function(listener) { var originalListener =  
listener; var view = this;  
  
// Create a new listener to wrap the original listener var newListener =  
Java.implement(Java.use('android.view.View$OnClickListener'), { onClick: function(v)  
{ console.log('[+] View clicked! Class: ' + view.getClassName() + ', ID: ' +  
view.getId()); if (originalListener) { originalListener.onClick(v); // Call the original click  
event } } });  
  
// Set the new listener this.setOnClickListener(newListener); }; });
```

[R33] Frida 常用脚本

R33: Frida 常用脚本速查手册

问题场景

你在使用 Frida 进行 Android 逆向时,经常遇到以下情况:

- 🌟 "我需要绕过 SSL Pinning 抓包,但不想从零写脚本"
- 🌟 "如何快速 Hook 所有 JNI 函数来分析 Native 层?"
- 🌟 "想拦截并修改网络请求,有现成的模板吗?"
- 🌟 "需要从 PC 端主动调用 App 的加密函数,怎么写 RPC?"
- 🌟 "App 检测到 Frida 就闪退,有通用的绕过脚本吗?"

本配方提供一套经过实战验证的 Frida 脚本模板库,按场景分类,可直接使用或快速修改。每个脚本都包含详细注释和使用说明。

工具清单

必需工具

- [x] Frida - 已安装并配置好 (参考 [Frida 使用指南](#))
- [x] Root 设备/模拟器 - 运行 Frida Server
- [x] 目标应用已安装 - 需要分析的 App

可选工具

- □ Python 3 - 用于 RPC 控制脚本
- □ mitmproxy/Burp Suite - 配合 SSL Pinning 绕过使用

-
- IDA Pro/Ghidra - 用于分析 Native 代码确定 Hook 点
-

前置条件

Frida 环境已配置并能成功 attach 到目标应用 了解基本的 JavaScript 语法 知道如何运行 Frida 脚本 (`frida -U -f com.app -l script.js`) 能识别需要 Hook 的类/函数名(至少知道包名)

脚本索引

本手册包含以下 8 类场景的脚本:

场景	脚本数量	适用情况
 绕过保护机制	3 个	反调试、反 Frida、SSL Pinning
网络拦截与修改	1 个	抓包、修改请求/响应
自动化 RPC 调用	1 套	主动调用加密函数、批量测试
JNI 函数分析	5 个	Native 层逆向、参数追踪
通用 Hook 模板	3 个	快速定位、批量 Hook
C 代码辅助工具	2 个	算法仿真、设备指纹生成

1. 绕过保护机制

脚本 1.1: 绕过 TracerPid 反调试检测

何时使用: App 通过读取 `/proc/self/status` 中的 `TracerPid` 来检测调试器。

工作原理: Hook `fgets` 函数,当检测到读取 `TracerPid` 时,将其值强制改为 0。

```
// bypass_tracerpid.js - Bypass TracerPid Anti-Debugging

// Step 1: Establish FILE* to path mapping
var fpMap = {};

// Hook fopen to record file paths
Interceptor.attach(Module.findExportByName(null, "fopen"), {
    onEnter: function (args) {
        this.path = args[0].readCString();
    },
    onLeave: function (retval) {
        if (!retval.isNull() && this.path) {
            fpMap[retval.toString()] = this.path;
            if (this.path.includes("/status")) {
                console.log("[+] fopen: " + this.path);
            }
        }
    },
});

// Hook fgets to modify TracerPid value
Interceptor.attach(Module.findExportByName(null, "fgets"), {
    onEnter: function (args) {
        this.buf = args[0];
        this.fp = args[2];
    },
    onLeave: function (retval) {
        if (retval.isNull()) return;

        var fp = this.fp.toString();
        var path = fpMap[fp];

        if (path && path.endsWith("/status")) {
            var line = this.buf.readCString();

            if (line && line.includes("TracerPid:")) {
                var oldValue = line.match(/TracerPid:\s*(\d+)/);
                this.buf.writeUtf8String("TracerPid:\t0\n");

                if (oldValue && oldValue[1] !== "0") {
                    console.log("\v [TracerPid] Modify: " + oldValue[1] + " -> 0");
                }
            }
        }
    },
});

console.log("[+] TracerPid Anti-Debugging Bypass activated");
```

使用方法:

```
# Spawn mode (recommended)
frida -U -f com.target.app -l bypass_tracerpid.js --no-pause

# Attach mode
frida -U com.target.app -l bypass_tracerpid.js
```

脚本 1.2: 隐藏 Frida 特征字符串

何时使用: App 通过搜索进程内存中的 "frida" 字符串来检测 Frida。

工作原理: Hook 字符串比较函数,当发现比较内容包含 "frida" 时,返回不匹配。

```
// hide_frida_strings.js - Hide Frida signature strings

// Hook strstr (most commonly used string search function)
var strstrPtr = Module.findExportByName("libc.so", "strstr");
if (strstrPtr) {
    Interceptor.attach(strstrPtr, {
        onEnter: function (args) {
            this.haystack = args[0].readCString();
            this.needle = args[1].readCString();
        },
        onLeave: function (retval) {
            if (this.haystack && this.needle) {
                var haystackLower = this.haystack.toLowerCase();
                var needleLower = this.needle.toLowerCase();

                if (haystackLower.includes("frida") || needleLower.includes("frida")) {
                    console.log("\u2713 [strstr] Intercept Frida Detection:");
                    console.log(
                        ' Search: "' +
                        this.needle +
                        '" in "' +
                        this.haystack.substring(0, 50) +
                        '..."'
                    );
                    retval.replace(ptr(0)); // Return NULL (not found)
                }
            }
        },
    });
    console.log("[+] strstr hook configured");
}

// Hook strcmp
var strcmpPtr = Module.findExportByName("libc.so", "strcmp");
if (strcmpPtr) {
    Interceptor.attach(strcmpPtr, {
        onEnter: function (args) {
            this.str1 = args[0].readCString();
            this.str2 = args[1].readCString();
        },
        onLeave: function (retval) {
            if (this.str1 && this.str2) {
                var str1Lower = this.str1.toLowerCase();
                var str2Lower = this.str2.toLowerCase();

                if (str1Lower.includes("frida") || str2Lower.includes("frida")) {
                    console.log("\u2713 [strcmp] Intercept Frida Detection:");
                    console.log(
                        ' Comparing: "' + this.str1 + '" vs "' + this.str2 + '"';
                    );
                    retval.replace(1); // Return non-zero (not equal)
                }
            }
        },
    });
}
```

```
        },
    },
);
console.log("[+] strcmp hook configured");
}

console.log("[+] Frida string hiding activated");
```

使用方法:

```
frida -U -f com.target.app -l hide_frida_strings.js --no-pause
```

脚本 1.3: 通用 SSL Pinning 绕过

何时使用: 需要使用中间人代理(如 Burp Suite) 抓取 HTTPS 流量,但 App 实现了证书校验。

工作原理: Hook 常见网络库(TrustManager、OkHttp3、HttpsURLConnection)的证书校验函数。

```
// bypass_ssl_pinning.js - Universal SSL Pinning Bypass Script

Java.perform(function () {
    console.log("[+] Starting SSL Pinning bypass...");

    // =====
    // 1. TrustManagerImpl (system level)
    // =====
    try {
        var TrustManagerImpl = Java.use(
            "com.android.org.conscrypt.TrustManagerImpl"
        );

        // android 7.0+
        TrustManagerImpl.verifyChain.implementation = function (
            untrustedChain,
            trustAnchorChain,
            host,
            clientAuth,
            ocspData,
            tlsSctData
        ) {
            console.log("✓ [TrustManagerImpl] Bypass cert validation: " + host);
            return untrustedChain; // Trust directly
        };

        console.log("[+] TrustManagerImpl Hook 成功");
    } catch (e) {
        console.log("[-] TrustManagerImpl not found: " + e);
    }

    // =====
    // 2. OkHttp3 (most commonly used)
    // =====
    try {
        var CertificatePinner = Java.use("okhttp3.CertificatePinner");

        CertificatePinner.check.overload(
            "java.lang.String",
            "java.util.List"
        ).implementation = function (hostname, peerCertificates) {
            console.log("✓ [OkHttp3] Bypass cert pinning: " + hostname);
            return; // Skip all checks
        };

        console.log("[+] OkHttp3 CertificatePinner Hook 成功");
    } catch (e) {
        console.log("[-] OkHttp3 not found: " + e);
    }

    // =====
    // 3. OkHttp3 - Hostname Verifier

```

```
// =====
try {
    var OkHostnameVerifier = Java.use(
        "okhttp3.internal.tls.OkHostnameVerifier"
    );

    OkHostnameVerifier.verify.overload(
        "java.lang.String",
        "javax.net.ssl.SSLSession"
    ).implementation = function (host, session) {
        console.log("✓ [OkHttp3] Bypass hostname validation: " + host);
        return true; // Always return validation passed
    };

    console.log("[+] OkHostnameVerifier Hook 成功");
} catch (e) {
    console.log("[-] OkHostnameVerifier not found: " + e);
}

// =====
// 4. HttpsURLConnection
// =====
try {
    var HttpsURLConnection = Java.use("javax.net.ssl.HttpsURLConnection");

    HttpsURLConnection.setDefaultHostnameVerifier.implementation = function (
        hostnameVerifier
    ) {
        console.log(
            "✓ [HttpsURLConnection] Intercept setDefaultHostnameVerifier"
        );
        return; // Don't set verifier
    };

    HttpsURLConnection.setSSLSocketFactory.implementation = function (
        socketFactory
    ) {
        console.log("✓ [HttpsURLConnection] Intercept setSSLSocketFactory");
        return; // Don't set factory
    };

    HttpsURLConnection.setHostnameVerifier.implementation = function (
        hostnameVerifier
    ) {
        console.log("✓ [HttpsURLConnection] Intercept setHostnameVerifier");
        return;
    };

    console.log("[+] HttpsURLConnection Hook 成功");
} catch (e) {
    console.log("[-] HttpsURLConnection Hook Failed: " + e);
}
```

```
// =====
// 5. SSLContext
// =====
try {
    var SSLContext = Java.use("javax.net.ssl.SSLContext");

    SSLContext.init.overload(
        "[Ljavax.net.ssl.KeyManager;",
        "[Ljavax.net.ssl.TrustManager;",
        "java.security.SecureRandom"
    ).implementation = function (km, tm, random) {
        console.log("\u2708 [SSLContext] Use custom TrustManager");

        // Create a TrustManager that trusts all certificates
        var TrustManager = Java.use("javax.net.ssl.X509TrustManager");
        var EmptyTrustManager = Java.registerClass({
            name: "com.frida.EmptyTrustManager",
            implements: [TrustManager],
            methods: {
                checkClientTrusted: function (chain, authType) {},
                checkServerTrusted: function (chain, authType) {},
                getAcceptedIssuers: function () {
                    return [];
                },
            },
        });
        var emptyTrustManager = EmptyTrustManager.$new();
        this.init(km, [emptyTrustManager], random);
    };

    console.log("[+] SSLContext Hook 成功");
} catch (e) {
    console.log("[-] SSLContext Hook Failed: " + e);
}

console.log("[+] SSL Pinning bypass configuration complete");
});
```

使用方法:

```
# 1. Start Burp Suite/mitmproxy on PC
# 2. Run script
frida -U -f com.target.app -l bypass_ssl_pinning.js --no-pause

# 3. View traffic in Burp/mitmproxy
```

2. 网络拦截与修改

脚本 2.1: OkHttp3 流量拦截与修改

何时使用: 需要在不使用代理的情况下,直接在 App 内部拦截和修改网络流量。

工作原理: Hook OkHttp3 的 `RealInterceptorChain.proceed` 方法,可以访问和修改请求/响应。

```
// intercept_okhttp.js - Intercept and modify OkHttp3 network requests

Java.perform(function () {
    console.log("[+] Starting OkHttp3 hook...");

    try {
        var RealInterceptorChain = Java.use(
            "okhttp3.internal.http.RealInterceptorChain"
        );

        RealInterceptorChain.proceed.implementation = function (request) {
            // =====
            // Request Interception
            // =====
            console.log("\n[REQUEST] =====");
            console.log("  URL: " + request.url().toString());
            console.log("  Method: " + request.method());

            // Print request headers
            var headers = request.headers();
            var headerCount = headers.size();
            if (headerCount > 0) {
                console.log("  Headers:");
                for (var i = 0; i < headerCount; i++) {
                    console.log("    " + headers.name(i) + ": " + headers.value(i));
                }
            }

            // Print request body
            var requestBody = request.body();
            if (requestBody) {
                try {
                    var Buffer = Java.use("okio.Buffer");
                    var buffer = Buffer.$new();
                    requestBody.writeTo(buffer);
                    var bodyString = buffer.readUtf8();
                    console.log("  Body: " + bodyString);
                } catch (e) {
                    console.log("  Body: [Cannot read]");
                }
            }
        }

        // =====
        // Modify Request (Optional)
        // =====
        var modifiedRequest = request
            .newBuilder()
            .header("X-Custom-Header", "Injected-By-Frida") // Add custom header
            .header("User-Agent", "FridaBot/1.0") // Modify User-Agent
            .build();

        // Execute request
    }
})
```

```
var response = this.proceed(modifiedRequest);

// =====
// Response Interception
// =====
console.log("\n[RESPONSE] =====");
console.log(" Code: " + response.code());
console.log(" Message: " + response.message());

// Print response headers
var respHeaders = response.headers();
var respHeaderCount = respHeaders.size();
if (respHeaderCount > 0) {
    console.log(" Headers:");
    for (var i = 0; i < respHeaderCount; i++) {
        console.log(
            "   " + respHeaders.name(i) + ": " + respHeaders.value(i)
        );
    }
}

// =====
// Modify Response (Optional)
// =====
var responseBody = response.body();
if (responseBody) {
    try {
        var contentType = responseBody.contentType();
        var bodyString = responseBody.string();

        console.log(" Body: " + bodyString.substring(0, 500));

        // Example: Modify JSON response field
        if (bodyString.includes('"status"')) {
            var modifiedBody = bodyString.replace(
                /"status":"error"/g,
                '"status":"success"'
            );
            console.log("✓ [Modify] Status field: error -> success");
        }

        // Rebuild response
        var MediaType = Java.use("okhttp3.MediaType");
        var ResponseBody = Java.use("okhttp3.ResponseBody");

        var newBody = ResponseBody.create(contentType, modifiedBody);

        return response.newBuilder().body(newBody).build();
    }

    // If not modified, need to recreate body (because it was already read)
    var ResponseBody = Java.use("okhttp3.ResponseBody");
    var newBody = ResponseBody.create(contentType, bodyString);
}
```

```
        return response.newBuilder().body(newBody).build();
    } catch (e) {
        console.log(" Body: [Read failed] " + e);
    }
}

return response;
};

console.log("[+] OkHttp3 Hook 成功");
} catch (e) {
    console.log("[-] Hook Failed: " + e);
}
});
});
```

使用方法:

```
frida -U -f com.target.app -l intercept_okhttp.js --no-pause
```

3. 自动化 RPC 调用

脚本 3.1: RPC 远程过程调用框架

何时使用: 需要从 PC 端批量调用 App 的加密函数、签名算法等,进行自动化测试。

Frida 脚本 (`rpc_agent.js`):

```
// rpc_agent.js - RPC export functions for Python calls

console.log("[+] RPC Agent loaded");

// Define exported RPC functions
rpc.exports = {
    // =====
    // Example 1: Call static encryption function
    // =====
    callEncrypt: function (plaintext) {
        var result = "";

        Java.perform(function () {
            try {
                // Modify to target app's actual class name and method name
                var CryptoUtil = Java.use("com.example.app.utils.CryptoUtil");

                // Call static method
                result = CryptoUtil.encrypt(plaintext);

                console.log('[RPC] encrypt("' + plaintext + "') = ' + result);
            } catch (e) {
                result = "ERROR: " + e;
                console.log("[-] " + result);
            }
        });
    },
    return result;
},
// =====
// Example 2: Call instance method
// =====
callInstanceMethod: function (className, methodName, args) {
    var result = "";

    Java.perform(function () {
        try {
            var TargetClass = Java.use(className);

            // Enumerate all instances
            Java.choose(className, {
                onMatch: function (instance) {
                    console.log("[RPC] Found instance: " + instance);

                    // Call instance method
                    result = instance[methodName].apply(instance, args);

                    console.log("[RPC] " + methodName + "() = " + result);
                },
                onComplete: function () {},
            });
        }
    });
}
```

```
        } catch (e) {
            result = "ERROR: " + e;
            console.log("[-] " + result);
        }
    });

    return result;
},

// =====
// Example 3: Call Native Function
// =====
callNativeFunction: function (libraryName, functionName, args) {
    try {
        var funcAddr = Module.findExportByName(libraryName, functionName);

        if (!funcAddr) {
            return "ERROR: Function not found";
        }

        // Define function signature (modify based on actual situation)
        // Example: int encrypt(char* input, char* output, int length)
        var nativeFunc = new NativeFunction(funcAddr, "int", [
            "pointer",
            "pointer",
            "int",
        ]);

        // Prepare parameters
        var input = Memory.allocUtf8String(args[0]);
        var output = Memory.alloc(1024);

        // Call function
        var ret = nativeFunc(input, output, args[0].length);

        var result = output.readCString();
        console.log("[RPC] Native " + functionName + "() returned: " + ret);
        console.log("[RPC] Output: " + result);

        return result;
    } catch (e) {
        return "ERROR: " + e;
    }
},
}

// =====
// Example 4: Get app info
// =====
getAppInfo: function () {
    var info = {};

    Java.perform(function () {
        var Context = Java.use("android.app.ActivityThread")
```

```
.currentApplication()
    .getApplicationContext();
var PackageManager = Context.getPackageManager();
var PackageName = Context.getPackageName();
var PackageInfo = PackageManager.getPackageInfo(PackageName, 0);

info.packageName = PackageName;
info.versionName = PackageInfo.versionName.value;
info.versionCode = PackageInfo.versionCode.value;

console.log("[RPC] App Info: " + JSON.stringify(info));
});

return info;
},
};

console.log("[+] RPC functions exported:");
console.log(" - callEncrypt(plaintext)");
console.log(" - callInstanceMethod(className, methodName, args)");
console.log(" - callNativeFunction(libraryName, functionName, args)");
console.log(" - getAppInfo");
```

Python 控制脚本 (`rpc_controller.py`):

```
# rpc_controller.py - Python RPC control script

import frida
import sys

def on_message(message, data):
    """Process messages from Frida script"""
    if message['type'] == 'send':
        print(f"[*] {message['payload']}")
    elif message['type'] == 'error':
        print(f"[!] Error: {message['stack']}")

def main():
    # =====
    # Connect to device and app
    # =====
    try:
        device = frida.get_usb_device(timeout=5)
        print(f"[+] Connected to device: {device}")
    except frida.TimedOutError:
        print("[-] Device connection timeout")
        sys.exit(1)

    # Attach to running app
    try:
        package_name = "com.example.app" # Modify to target app package name
        session = device.attach(package_name)
        print(f"[+] Attached to: {package_name}")
    except frida.ProcessNotFoundError:
        print(f"[-] Process not found: {package_name}")
        print("[*] Please ensure app is running")
        sys.exit(1)

    # =====
    # Load Frida Script
    # =====
    with open("rpc_agent.js", "r", encoding="utf-8") as f:
        script_code = f.read()

    script = session.create_script(script_code)
    script.on('message', on_message)
    script.load()
    print("[+] Frida script loaded\n")

    # =====
    # Get RPC API
    # =====
    api = script.exports

    # =====
    # Example 1: Call encryption function
    # =====
```

```
print("==" * 60)
print("Example 1: Call encryption function")
print("==" * 60)

test_data = "Hello, Frida RPC!"
encrypted = api.call_encrypt(test_data)
print(f"Plaintext: {test_data}")
print(f"Ciphertext: {encrypted}\n")

# =====
# Example 2: Batch test
# =====
print("==" * 60)
print("Example 2: Batch test")
print("==" * 60)

test_cases = [
    "test1",
    "test2",
    "test3",
    "a" * 100, # Long string
    "", # Empty string
]

for i, test_input in enumerate(test_cases):
    result = api.call_encrypt(test_input)
    print(f"[{i+1}] {test_input[:20]}:<20} -> {result}")

print()

# =====
# Example 3: Get app info
# =====
print("==" * 60)
print("Example 3: Get app info")
print("==" * 60)

app_info = api.get_app_info()
print(f"Package name: {app_info['packageName']}")
print(f"Version: {app_info['versionName']} ({app_info['versionCode']})")

# =====
# Keep session alive
# =====
print("\n[+] RPC session established, press Ctrl+C to exit")
try:
    sys.stdin.read()
except KeyboardInterrupt:
    print("\n[*] Disconnecting...")

session.detach()
print("[+] Disconnected")
```

```
if __name__ == "__main__":
    main()
```

使用方法:

```
# 1. Start the app on device first
# 2. Run Python script
python3 rpc_controller.py

# Output example:
# [+] Connected to device: ...
# [+] Attached to: com.example.app
# [+] Frida script loaded
#
# Plaintext: Hello, Frida RPC!
# Ciphertext: SGVsbG8sIEZyaWRhIFJQQyE=
```

4. JNI 函数分析

脚本 4.1: 枚举 SO 文件中的所有 JNI 函数

何时使用: 需要找出某个 Native 库导出了哪些 JNI 函数。

工作原理: 扫描 SO 文件的导出表,过滤出所有以 `Java_` 开头的符号。

```
// enumerate_jni.js - Enumerate all JNI functions in specified SO file

function enumerateJNIFunctions(libraryName) {
    var module = Process.findModuleByName(libraryName);

    if (!module) {
        console.log("[-] Module not found: " + libraryName);
        console.log("[*] Trying to wait for module loading...");

        // Monitor dlopen
        Interceptor.attach(Module.findExportByName(null, "dlopen"), {
            onEnter: function (args) {
                var path = args[0].readCString();
                if (path && path.includes(libraryName)) {
                    console.log("[+] Detected target library loading: " + path);
                    this.target = true;
                }
            },
            onLeave: function (retval) {
                if (this.target && !retval.isNull()) {
                    setTimeout(function () {
                        enumerateJNIFunctions(libraryName);
                    }, 500);
                }
            },
        });
    }

    return;
}

console.log("\n" + "=" .repeat(70));
console.log("  JNI Function Enumeration: " + libraryName);
console.log("  Base address: " + module.base);
console.log("  Size: " + (module.size / 1024).toFixed(2) + " KB");
console.log("=".repeat(70) + "\n");

var exports = module.enumerateExports();
var jniFunctions = [];

// Filter JNI functions
exports.forEach(function (exp) {
    if (exp.name.startsWith("Java_")) {
        jniFunctions.push(exp);
    }
});

if (jniFunctions.length === 0) {
    console.log("[-] No JNI functions found");
    return;
}

// Sort by name
```

```
jniFunctions.sort(function (a, b) {
    return a.name.localeCompare(b.name);
});

// Print results
jniFunctions.forEach(function (exp, index) {
    console.log("[ " + index + " ] " + exp.name);
    console.log("  Address: " + exp.address);
    console.log("  Offset: +" + ptr(exp.address).sub(module.base));

    // Parse JNI function name
    // Format: Java_PackageName_ClassName_MethodName
    var parts = exp.name.split("_");
    if (parts.length >= 4) {
        var packageAndClass = parts.slice(1, -1).join(".");
        var methodName = parts[parts.length - 1];
        console.log(
            "  Java Method: " + packageAndClass + "." + methodName + "()"
        );
    }
    console.log();
});

console.log("[+] Found " + jniFunctions.length + " JNI functions\n");
}

// =====
// Usage example
// =====
var TARGET_LIBRARY = "libnative-lib.so"; // Modify to target SO file name

// Method 1: If library is already loaded
enumerateJNIFunctions(TARGET_LIBRARY);

// Method 2: Wait for library to load then enumerate
// (If not found above, will automatically enable monitoring)
```

使用方法:

```
frida -U -f com.target.app -l enumerate_jni.js --no-pause
```

脚本 4.2: Hook 单个 JNI 函数

何时使用: 已知具体的 JNI 函数名, 需要追踪其参数和返回值。

工作原理: 通过函数名直接定位并 Hook, 解析 JNIEnv 指针和参数。

```
// hook_jni_function.js - Hook single JNI function

function hookJNIFunction(libraryName, functionName) {
    var funcAddr = Module.findExportByName(libraryName, functionName);

    if (!funcAddr) {
        console.log("[-] Function not found: " + functionName);
        return;
    }

    console.log("[+] Hooking: " + functionName);
    console.log("  Address: " + funcAddr);

    Interceptor.attach(funcAddr, {
        onEnter: function (args) {
            console.log("\n" + "=" .repeat(60));
            console.log("[JNI CALL] " + functionName);
            console.log("=".repeat(60));
            console.log("  JNIEnv*: " + args[0]);
            console.log("  jobject/jclass: " + args[1]);

            // Try to parse parameters (starting from args[2])
            for (var i = 2; i < 8 && i < args.length; i++) {
                var arg = args[i];
                console.log("  arg[" + (i - 2) + "]: " + arg);

                if (arg.isNull()) {
                    console.log("    -> null");
                    continue;
                }

                // Try to parse as jstring
                try {
                    var env = Java.vm.getEnv();
                    var strPtr = env.getStringUtfChars(arg, null);
                    var str = strPtr.readCString();

                    if (str && str.length > 0 && str.length < 500) {
                        console.log('    -> jstring: "' + str + '"');
                    }

                    env.releaseStringUtfChars(arg, strPtr);
                    continue;
                } catch (e) {}

                // Try to parse as integer
                try {
                    var intValue = arg.toInt32();
                    console.log(
                        "    -> jint: " + intValue + " (0x" + intValue.toString(16) + ")"
                    );
                    continue;
                }
            }
        }
    });
}
```

```
        } catch (e) {}

        // Try to parse as byte array
        try {
            var env = Java.vm.getEnv();
            var arrayLen = env.getArrayLength(arg);

            if (arrayLen > 0 && arrayLen < 1024) {
                console.log("    -> jbyteArray[" + arrayLen + "]");

                var bytePtr = env.getByteArrayElements(arg, null);
                var bytes = bytePtr.readByteArray(Math.min(arrayLen, 64));
                console.log(
                    hexdump(bytes, {
                        offset: 0,
                        length: Math.min(arrayLen, 64),
                        header: false,
                        ansi: false,
                    })
                );
                env.releaseByteArrayElements(arg, bytePtr, 0);
            }
            continue;
        } catch (e) {}

        console.log("    -> Pointer: " + arg);
    }
},
onLeave: function (retval) {
    console.log("\n  [Return Value]: " + retval);

    if (retval.isNull()) {
        console.log("    -> null");
        return;
    }

    // Try to parse return value
    try {
        var env = Java.vm.getEnv();
        var strPtr = env.getStringUtfChars(retval, null);
        var str = strPtr.readCString();

        if (str && str.length > 0 && str.length < 500) {
            console.log('    -> jstring: "' + str + '"');
        }

        env.releaseStringUtfChars(retval, strPtr);
    } catch (e) {
        try {
            var intValue = retval.toInt32();
            console.log("    -> jint: " + intValue);
        } catch (e2) {
    }
}
```

```
        console.log("    -> Pointer: " + retval);
    }
}

console.log("=".repeat(60) + "\n");
},
});

console.log("[+] Hook configured\n");
}

// =====
// Usage example
// =====
hookJNIFunction("libnative-lib.so", "Java_com_example_app_Crypto_encrypt");
```

使用方法:

```
frida -U -f com.target.app -l hook_jni_function.js --no-pause
```

脚本 4.3: 批量 Hook 所有 JNI 函数

何时使用: 不确定哪个 JNI 函数与目标功能相关,需要全部拦截观察。

工作原理: 枚举所有 JNI 函数,批量设置 Hook。

```
// hook_all_jni.js - Batch hook all JNI functions

function hookAllJNI(libraryName) {
    var module = Process.findModuleByName(libraryName);

    if (!module) {
        console.log("[-] Module not found, waiting for load: " + libraryName);

        Interceptor.attach(Module.findExportByName(null, "dlopen"), {
            onEnter: function (args) {
                var path = args[0].readCString();
                if (path && path.includes(libraryName)) {
                    this.target = true;
                }
            },
            onLeave: function (retval) {
                if (this.target && !retval.isNull()) {
                    setTimeout(function () {
                        hookAllJNI(libraryName);
                    }, 500);
                }
            },
        });
    }

    return;
}

console.log("[+] Starting batch JNI function hook: " + libraryName);

var exports = module.enumerateExports();
var hookedCount = 0;

exports.forEach(function (exp) {
    if (!exp.name.startsWith("Java_")) {
        return;
    }

    try {
        Interceptor.attach(exp.address, {
            onEnter: function (args) {
                console.log("\n[JNI] " + exp.name);

                // Simplified output, only print first 3 parameters
                for (var i = 0; i < 5 && i < args.length; i++) {
                    var arg = args[i];

                    if (i === 0) {
                        console.log(" JNIEnv*: " + arg);
                    } else if (i === 1) {
                        console.log(" jobject: " + arg);
                    } else {
                        console.log(" arg[" + (i - 2) + "]: " + arg);
                    }
                }
            }
        });
    }
});
```

```
// Try to parse string
if (!arg.isNull()) {
    try {
        var env = Java.vm.getEnv();
        var str = env.getStringUtfChars(arg, null).readCString();
        if (str && str.length > 0 && str.length < 100) {
            console.log('    -> "' + str + '"');
        }
        env.releaseStringUtfChars(arg, str);
    } catch (e) {}
}
},
};

onLeave: function (retval) {
    console.log("  Return: " + retval);
},
});

hookedCount++;
} catch (e) {
    console.log("[-] Hook failed: " + exp.name);
}
});

console.log("[+] Successfully hooked " + hookedCount + " JNI functions");
}

// =====
// Usage
// =====
hookAllJNI("libnative-lib.so");
```

使用方法:

```
frida -U -f com.target.app -l hook_all_jni.js --no-pause
```

脚本 4.4: Hook JNI_OnLoad 函数

何时使用: 需要在 Native 库加载时的初始化阶段进行分析。

工作原理: Hook `JNI_OnLoad`, 这是 Native 库加载时系统调用的第一个函数。

```
// hook_jni_onload.js - Hook JNI_OnLoad function

function hookJNIOnLoad(libraryName) {
    var onLoadAddr = Module.findExportByName(libraryName, "JNI_OnLoad");

    if (!onLoadAddr) {
        console.log("[-] JNI_OnLoad not found: " + libraryName);
        return;
    }

    console.log("[+] Hooking JNI_OnLoad");
    console.log("  Address: " + onLoadAddr);

    Interceptor.attach(onLoadAddr, {
        onEnter: function (args) {
            console.log("\n" + "=" .repeat(60));
            console.log("[JNI_OnLoad] Called");
            console.log("=".repeat(60));
            console.log("  JavaVM*: " + args[0]);
            console.log("  reserved: " + args[1]);

            this.vm = args[0];
        },
        onLeave: function (retval) {
            var jniVersion = retval.toInt32();
            console.log("  Return JNI Version: " + jniVersion);

            // Parse version number
            var major = (jniVersion >> 16) & 0xffff;
            var minor = jniVersion & 0xffff;
            console.log("  -> JNI_VERSION_" + major + "_" + minor);

            console.log("=".repeat(60));
        }
    });

    // After JNI_OnLoad completes, can start hooking other JNI functions
    setTimeout(function () {
        console.log(
            "\n[+] JNI_OnLoad completed, starting to hook JNI functions...\n"
        );
        // Can call other hook functions here
        }, 100);
    },
    });
}

// Monitor library loading
Interceptor.attach(Module.findExportByName(null, "dlopen"), {
    onEnter: function (args) {
        var path = args[0].readCString();
        console.log("[dlopen] " + path);
```

```
if (path && path.includes("libnative-lib.so")) {
    console.log("[+] Detected target library loading");
    this.target = true;
}
},
onLeave: function (retval) {
    if (this.target && !retval.isNull()) {
        setTimeout(function () {
            hookJNIOnLoad("libnative-lib.so");
        }, 100);
    }
},
});
console.log("[+] Monitoring library loading...");
```

使用方法:

```
frida -U -f com.target.app -l hook_jni_onload.js --no-pause
```

5. 通用 Hook 模板

脚本 5.1: Hook Java 方法(支持重载)

何时使用: 需要拦截某个 Java 类的特定方法。

工作原理: 使用 `Java.use()` 加载类,然后替换方法实现。

```
// hook_java_method.js - Universal Java method hook template

function hookJavaMethod(className, methodName) {
    Java.perform(function () {
        try {
            var targetClass = Java.use(className);

            // Get all overloads
            var overloads = targetClass[methodName].overloads;

            console.log(
                "[+] Found " +
                overloads.length +
                " overloads: " +
                className +
                "." +
                methodName
            );
        }

        // Hook all overloads
        overloads.forEach(function (overload) {
            overload.implementation = function () {
                console.log("\n[CALL] " + className + "." + methodName);

                // Print parameters
                for (var i = 0; i < arguments.length; i++) {
                    console.log(" arg[" + i + "]: " + arguments[i]);
                }

                // Call original method
                var result = this[methodName].apply(this, arguments);

                console.log(" Return: " + result);

                return result;
            };
        });
    });

    console.log("[+] Hook complete");
} catch (e) {
    console.log("[-] Hook failed: " + e);
}
});

// Usage example
hookJavaMethod("android.util.Log", "d");
hookJavaMethod("com.example.app.Crypto", "encrypt");
```

使用方法:

```
frida -U -f com.target.app -l hook_java_method.js --no-pause
```

脚本 5.2: Hook 类的所有方法

何时使用: 需要观察某个类的所有方法调用情况。

工作原理: 使用反射获取类的所有方法,批量 Hook。

```
// hook_all_methods.js - Hook all methods of a class

function hookAllMethods(className) {
    Java.perform(function () {
        try {
            var targetClass = Java.use(className);
            var methods = targetClass.class.getDeclaredMethods();

            console.log("[+] Class: " + className);
            console.log("[+] Found " + methods.length + " methods\n");

            var hookedCount = 0;

            methods.forEach(function (method) {
                try {
                    var methodName = method.getName();

                    // Skip certain methods
                    if (methodName === "toString" || methodName === "hashCode") {
                        return;
                    }

                    var overloads = targetClass[methodName].overloads;

                    overloads.forEach(function (overload) {
                        overload.implementation = function () {
                            console.log("\n[" + className + "] " + methodName + "()");
                            if (arguments.length > 0) {
                                console.log(" Parameters:");
                                for (var i = 0; i < arguments.length; i++) {
                                    console.log(" [" + i + "] " + arguments[i]);
                                }
                            }
                        }

                        var result = this[methodName].apply(this, arguments);

                        console.log(" Return: " + result);

                        return result;
                    });
                };
            });

            hookedCount++;
        } catch (e) {
            // Some methods may not be hookable
        }
    });

    console.log("[+] Successfully hooked " + hookedCount + " methods");
} catch (e) {
    console.log("[+] Failed: " + e);
}
```

```
        }  
    } );  
  
// Usage  
hookAllMethods("com.example.app.utils.CryptoUtil");
```

使用方法:

```
frida -U -f com.target.app -l hook_all_methods.js --no-pause
```

脚本 5.3: Hook 构造函数

何时使用: 需要监控对象创建时机和构造参数。

工作原理: Hook 类的 `$init` 方法(Frida 中构造函数的特殊名称)。

```
// hook_constructor.js - Hook class constructor

function hookConstructor(className) {
    Java.perform(function () {
        try {
            var targetClass = Java.use(className);

            // $init is the special name for constructors
            var overloads = targetClass.$init.overloads;

            console.log(
                "[+] Found " + overloads.length + " constructors: " + className
            );

            overloads.forEach(function (overload) {
                overload.implementation = function () {
                    console.log("\n[NEW] " + className + "()");

                    if (arguments.length > 0) {
                        console.log(" Constructor parameters:");
                        for (var i = 0; i < arguments.length; i++) {
                            console.log("    [" + i + "] " + arguments[i]);
                        }
                    }
                }

                // Call original constructor
                var result = this.$init.apply(this, arguments);

                console.log(" Instance: " + this);

                return result;
            });
        } catch (e) {
            console.log("[-] Failed: " + e);
        }
    });
}

// Usage
hookConstructor("javax.crypto.spec.SecretKeySpec");
```

使用方法:

```
frida -U -f com.target.app -l hook_constructor.js --no-pause
```

6. C 代码辅助工具

工具 6.1: 算法仿真工具

何时使用: 在 IDA/Ghidra 中看到加密/解密算法逻辑,需要提取出来独立验证。

示例: XOR 加密算法仿真

```
// emulate_xor_encrypt.c - Emulate XOR encryption algorithm

#include <stdio.h>
#include <string.h>
#include <stdint.h>

// Algorithm extracted from IDA pseudocode
void encrypt_data(uint8_t* data, size_t len, uint8_t key) {
    for (size_t i = 0; i < len; i++) {
        data[i] = (data[i] ^ key) + 5;
    }
}

// Corresponding decryption algorithm
void decrypt_data(uint8_t* data, size_t len, uint8_t key) {
    for (size_t i = 0; i < len; i++) {
        data[i] = (data[i] - 5) ^ key;
    }
}

// Helper function: Print hexadecimal
void print_hex(const char* label, uint8_t* data, size_t len) {
    printf("%s: ", label);
    for (size_t i = 0; i < len; i++) {
        printf("%02x ", data[i]);
    }
    printf("\n");
}

int main() {
    // Test data
    uint8_t plaintext[] = "Hello, Android Reverse Engineering!";
    size_t len = strlen((char*)plaintext);
    uint8_t key = 0x5A;

    printf("==== XOR Encryption Algorithm Test ====\n\n");

    // Plaintext
    printf("Plaintext: %s\n", plaintext);
    print_hex("Plaintext HEX", plaintext, len);
    printf("\n");

    // Encrypt
    encrypt_data(plaintext, len, key);
    printf("After encryption:\n");
    print_hex("Ciphertext HEX", plaintext, len);
    printf("\n");

    // Decrypt
    decrypt_data(plaintext, len, key);
    printf("After decryption: %s\n", plaintext);
    print_hex("Decrypted HEX", plaintext, len);
}
```

```
    return 0;  
}
```

编译和运行:

```
# Compile the program  
gcc emulate_xor_encrypt.c -o emulate  
  
# Run the program  
.emulate  
  
# Output:  
# === XOR Encryption Algorithm Test ===  
#  
# Plaintext: Hello, Android Reverse Engineering!  
# Plaintext HEX: 48 65 6c 6c 6f 2c 20 41 6e 64 72 6f 69 64 ...  
#  
# After encryption:  
# Ciphertext HEX: 17 30 39 39 32 79 75 16 39 31 2d 32 36 31 ...  
#  
# After decryption: Hello, Android Reverse Engineering!
```

工具 6.2: 设备指纹生成工具

何时使用: 需要批量生成虚拟设备的指纹信息用于测试。

示例: 设备指纹生成

```
// device_fingerprint.c - 设备指纹生成工具

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// Execute shell command and get output
char* execute_command(const char* cmd) {
    FILE* fp = popen(cmd, "r");
    if (!fp) return NULL;

    char* result = malloc(256);
    if (fgets(result, 256, fp) == NULL) {
        free(result);
        pclose(fp);
        return NULL;
    }

    // Remove newline character
    result[strcspn(result, "\n")] = 0;
    pclose(fp);

    return result;
}

// Get system property
char* get_prop(const char* key) {
    char command[256];
    snprintf(command, sizeof(command), "getprop %s", key);
    return execute_command(command);
}

int main() {
    printf("{\n");
    printf("  \"timestamp\": %ld,\n", time(NULL));
    printf("  \"device\": {\n");

    // Device info
    const char* props[] = {
        "ro.product.brand",
        "ro.product.model",
        "ro.product.manufacturer",
        "ro.product.device",
        "ro.build.version.release",
        "ro.build.version.sdk",
        "ro.build.fingerprint",
        "ro.serialno",
        "ro.boot.serialno"
    };

    int num_props = sizeof(props) / sizeof(props[0]);
```

```
for (int i = 0; i < num_props; i++) {
    char* value = get_prop(props[i]);

    if (value) {
        // Extract last part of property name
        const char* last_dot = strrchr(props[i], '.');
        const char* key = last_dot ? last_dot + 1 : props[i];

        printf("    \"%s\": \"%s\"", key, value);

        if (i < num_props - 1) {
            printf(",");
        }
        printf("\n");

        free(value);
    }
}

printf(" }\n");
printf("}\n");

return 0;
}
```

编译和使用:

```
# Push to device and compile
adb push device_fingerprint.c /data/local/tmp/
adb shell
cd /data/local/tmp
gcc device_fingerprint.c -o fingerprint
chmod +x fingerprint

# Run the program
./fingerprint

# Output JSON format device fingerprint:
# {
#   "timestamp": 1734518400,
#   "device": {
#     "brand": "google",
#     "model": "Pixel 5",
#     "manufacturer": "Google",
#     ...
#   }
# }

# Save to file
./fingerprint > /sdcard/device_info.json
```

常见问题排查

问题 1: Hook 没有生效

可能原因:

1. Hook 时机太晚,目标函数已经执行完毕
2. 类名或方法名拼写错误
3. 使用了 Attach 模式,但 DEX 还未加载

解决方案:

```
# 1. Use Spawn mode (Recommended)
frida -U -f com.target.app -l script.js --no-pause

# 2. Check if class name is correct
```

```
Java.perform(function () {
    Java.enumerateLoadedClasses({
        onMatch: function (className) {
            if (className.indexOf("Crypto") !== -1) {
                console.log("[+] Found class: " + className);
            }
        },
        onComplete: function () {},
    });
});
```

```
# 3. Delayed hook (if using Attach mode)
```

```
setTimeout(function () {
    hookJavaMethod("com.example.app.Crypto", "encrypt");
}, 2000);
```

问题 2: Hook JNI 函数时 App 崩溃

可能原因:

1. 读取了无效的指针
2. JNIEnv 使用不当
3. 字节数组释放问题

解决方案:

```
// Add try-catch protection
Interceptor.attach(funcAddr, {
    onEnter: function (args) {
        try {
            // Check pointer validity first
            if (!args[2].isNull()) {
                var env = Java.vm.getEnv();
                // ... Process parameters
            }
        } catch (e) {
            console.log("[-] Caught exception: " + e);
            // Don't re-throw to avoid crash
        }
    },
});
```

问题 3: RPC 调用时提示类不存在

可能原因:

1. 类还未加载到内存
2. 类名错误或被混淆
3. 使用了动态加载的 DEX

解决方案:

```
# On Python side, wait for class to load first
api.wait_for_class("com.example.app.Crypto") # Custom wait function
```

```
// Or check in Frida script
rpc.exports = {
  callEncrypt: function (input) {
    var result = "";

    Java.perform(function () {
      // Check if class exists first
      try {
        var Crypto = Java.use("com.example.app.Crypto");
        result = Crypto.encrypt(input);
      } catch (e) {
        // Try to enumerate and find
        Java.enumerateLoadedClasses({
          onMatch: function (className) {
            if (className.includes("Crypto")) {
              console.log("[+] Found: " + className);
            }
          },
          onComplete: function () {},
        });
      }

      result = "ERROR: " + e;
    });
  },
};

return result;
},
};
```

问题 4: SSL Pinning 绕过失败

可能原因:

1. 应用使用了自定义的 SSL Pinning 实现
2. Native 层实现的 Pinning
3. 使用了第三方网络库(如 Cronet)

解决方案:

```
// 1. Add more hook points
Java.perform(function () {
    // Hook custom TrustManager
    Java.enumerateLoadedClasses({
        onMatch: function (className) {
            if (
                className.includes("TrustManager") ||
                className.includes("Certificate")
            ) {
                console.log("[+] Found suspicious class: " + className);

                try {
                    var clazz = Java.use(className);
                    var methods = clazz.class.getDeclaredMethods();

                    methods.forEach(function (method) {
                        var methodName = method.getName();
                        if (methodName.includes("check") || methodName.includes("verify")) {
                            console.log("[+] Hook: " + className + "." + methodName);
                            // Batch hook
                        }
                    });
                } catch (e) {}
            }
        },
        onComplete: function () {},
    });
});

// 2. Hook native layer SSL_CTX_set_verify
var SSL_CTX_set_verify = Module.findExportByName(
    "libssl.so",
    "SSL_CTX_set_verify"
);
if (SSL_CTX_set_verify) {
    Interceptor.attach(SSL_CTX_set_verify, {
        onEnter: function (args) {
            console.log("\u2713 [SSL_CTX_set_verify] Bypass");
            args[1] = ptr(0); // SSL_VERIFY_NONE
        },
    });
}
```

相关资源

场景延伸

- [Recipe: 抓包分析 Android 应用的网络流量](#) - 配合 SSL Pinning 绕过使用
- [Recipe: 分析并提取 Android 应用的加密密钥](#) - 密码学分析的完整流程

工具深入

- [Frida 使用指南](#) - 完整的 Frida 使用手册
- [Frida 内部原理](#) - 深入理解 Frida 工作机制

案例分析

- [案例: 音乐 App 分析](#) - 综合运用多个脚本

参考资料

- [JNI 函数速查](#)
- [常见加密算法识别](#)

快速参考

脚本速查表

需求	使用脚本	难度
绕过 TracerPid 检测	bypass_tracerpid.js	★
隐藏 Frida 字符串	hide_frida_strings.js	★
绕过 SSL Pinning	bypass_ssl_pinning.js	★
拦截网络请求	intercept_okhttp.js	★★
RPC 调用加密函数	rpc_agent.js + rpc_controller.py	★★
枚举 JNI 函数	enumerate_jni.js	★
Hook JNI 函数	hook_jni_function.js	★★
批量 Hook JNI	hook_all_jni.js	★★
Hook 构造函数	hook_constructor.js	★

常用命令

```
# 1. Spawn mode run script (recommended)
frida -U -f com.target.app -l script.js --no-pause

# 2. Attach mode
frida -U com.target.app -l script.js

# 3. List all processes
frida-ps -Ua

# 4. Interactive REPL
frida -U com.target.app

# 5. Load multiple scripts
frida -U -f com.target.app -l script1.js -l script2.js --no-pause

# 6. Export output to file
frida -U com.target.app -l script.js > output.log 2>&1
```

最佳实践

1. 优先使用 Spawn 模式 - 确保在应用启动前 Hook 就绪
2. 添加异常处理 - 防止脚本错误导致应用崩溃
3. 适度打印日志 - 过多日志会影响性能
4. 模块化组织 - 将常用函数封装为独立模块
5. 保存脚本库 - 建立自己的脚本模板库

 提示: 这些脚本都是模板,实际使用时需要根据目标 App 的具体情况迸行调整。建议先理解脚本原理,再修改关键参数(如类名、方法名、SO 文件名等)。

[R34] 自动化脚本

R34: 自动化脚本 (Automation Scripts)

在 Android 逆向工程中，自动化脚本可以极大地提高效率，例如自动安装 APK、重启应用、模拟点击以及批量处理设备。Python 是编写这些脚本的首选语言。

1. 基础 ADB 封装 (Python)

虽然可以直接在 shell 中运行 `adb` 命令，但在 Python 中封装一层可以更方便地进行逻辑控制。

```
import subprocess
import time
import os

class AdbWrapper:
    def __init__(self, device_id=None):
        self.device_id = device_id

    def run_cmd(self, cmd):
        adb_cmd = ["adb"]
        if self.device_id:
            adb_cmd.extend(["-s", self.device_id])
        adb_cmd.extend(cmd)

        try:
            result = subprocess.run(
                adb_cmd,
                capture_output=True,
                text=True,
                check=True
            )
            return result.stdout.strip()
        except subprocess.CalledProcessError as e:
            print(f"Error running command {' '.join(adb_cmd)}: {e.stderr}")
            return None

    def install(self, apk_path):
        print(f"Installing {apk_path}...")
        return self.run_cmd(["install", "-r", apk_path])

    def uninstall(self, package_name):
        print(f"Uninstalling {package_name}...")
        return self.run_cmd(["uninstall", package_name])

    def start_app(self, package_name, activity_name):
        print(f"Starting {package_name}/{activity_name}...")
        return self.run_cmd(["shell", "am", "start", "-n", f"{package_name}/"
{activity_name}"])

    def stop_app(self, package_name):
        print(f"Stopping {package_name}...")
        return self.run_cmd(["shell", "am", "force-stop", package_name])

    def clear_data(self, package_name):
        print(f"Clearing data for {package_name}...")
        return self.run_cmd(["shell", "pm", "clear", package_name])

    def click(self, x, y):
        return self.run_cmd(["shell", "input", "tap", str(x), str(y)])

    def swipe(self, x1, y1, x2, y2, duration=500):
        return self.run_cmd(["shell", "input", "swipe", str(x1), str(y1), str(x2),
```

```
str(y2), str(duration))]

def input_text(self, text):
    # Note: special characters might need escaping
    return self.run_cmd(["shell", "input", "text", text])

def screenshot(self, remote_path="/sdcard/screenshot.png",
local_path="screenshot.png"):
    self.run_cmd(["shell", "screencap", "-p", remote_path])
    self.run_cmd(["pull", remote_path, local_path])
    print(f"Screenshot saved to {local_path}")

# 使用示例
if __name__ == "__main__":
    adb = AdbWrapper() # 默认连接第一个设备

    # 打印连接的设备
    print("Devices:", adb.run_cmd(["devices"]))

    # adb.start_app("com.example.app", "com.example.app.MainActivity")
    # time.sleep(5)
    # adb.click(500, 1000)
    # adb.screenshot()
```

2. UIAutomator2 自动化

`uiautomator2` 是一个更强大的 UI 自动化库，可以更方便地进行元素定位和操作。

安装

```
pip install uiautomator2
```

使用示例

```
import uiautomator2 as u2
import time

# 连接设备 (USB)
d = u2.connect()
# 或通过 WiFi: d = u2.connect('192.168.1.100')

print(f"Connected to device: {d.info}")

# 启动 App
pkg_name = "com.example.android.apis"
d.app_start(pkg_name)

# 等待 App 启动
d.wait_activity(".ApiDemos", timeout=10)

# 查找并点击元素 (支持多种选择器)
try:
    # 通过文本查找并点击
    if d(text="App").exists:
        d(text="App").click()

    # 通过 resourceId 查找
    # d(resourceId="com.example:id/button").click()

    # 滚动查找 (向下滑动直到找到文本为 'Notification' 元素)
    d(scrollable=True).scroll.to(text="Notification")
    d(text="Notification").click()

    # 输入文本
    # d(resourceId="com.example:id/edit_text").set_text("Hello World")

    # 截图
    d.screenshot("uiauto_screenshot.jpg")

except Exception as e:
    print(f"Error: {e}")

finally:
    # 停止 App
    # d.app_stop(pkg_name)
    pass
```

3. 批量管理工具

用于批量安装 APK、设置代理等操作。

```
import os
import glob
from concurrent.futures import ThreadPoolExecutor

class BatchManager:
    def __init__(self, adb_wrapper):
        self.adb = adb_wrapper

    def install_all(self, directory):
        """批量安装目录下所有 APK"""
        apk_files = glob.glob(os.path.join(directory, "*.apk"))
        print(f"Found {len(apk_files)} APKs.")

        # 使用线程池并发安装 (注意: ADB 并发可能不稳定, 视情况调整)
        with ThreadPoolExecutor(max_workers=3) as executor:
            executor.map(self.adb.install, apk_files)

    def setup_proxy(self, host, port):
        """设置全局 HTTP 代理"""
        print(f"Setting global http proxy to {host}:{port}...")
        self.adb.run_cmd(["shell", "settings", "put", "global", "http_proxy",
                         f"{host}:{port}"])

    def clear_proxy(self):
        """清除全局 HTTP 代理"""
        print("Clearing global http proxy...")
        self.adb.run_cmd(["shell", "settings", "put", "global", "http_proxy", ":0"])

    def get_device_info(self):
        """获取设备信息"""
        info = {}
        info['brand'] = self.adb.run_cmd(["shell", "getprop", "ro.product.brand"])
        info['model'] = self.adb.run_cmd(["shell", "getprop", "ro.product.model"])
        info['sdk'] = self.adb.run_cmd(["shell", "getprop", "ro.build.version.sdk"])
        info['android'] = self.adb.run_cmd(["shell", "getprop",
                                             "ro.build.version.release"])
        return info

    # 使用示例
if __name__ == "__main__":
    adb = AdbWrapper()
    manager = BatchManager(adb)

    # 批量安装当前目录下 apks 文件夹中所有 apk
    # manager.install_all("./apks")

    # 设置代理以便抓包
    # manager.setup_proxy("192.168.1.10", "8080")

    # 获取设备信息
    # info = manager.get_device_info()
    # print(info)
```

4. Frida 自动化脚本

结合 Frida 进行自动化 Hook。

```
import frida
import sys
import time

def on_message(message, data):
    if message['type'] == 'send':
        print(f"[*] {message['payload']}")
    elif message['type'] == 'error':
        print(f"[!] {message['stack']}")

def run_frida_script(package_name, script_path):
    """运行 Frida 脚本"""
    try:
        # 连接设备
        device = frida.get_usb_device(timeout=5)
        print(f"[+] Connected to: {device}")

        # 启动或附加到应用
        try:
            pid = device.spawn([package_name])
            session = device.attach(pid)
            spawned = True
        except:
            session = device.attach(package_name)
            spawned = False

        # 加载脚本
        with open(script_path, 'r') as f:
            script_code = f.read()

        script = session.create_script(script_code)
        script.on('message', on_message)
        script.load()

        # 如果是 spawn 模式, 恢复应用
        if spawned:
            device.resume(pid)

        print(f"[+] Script loaded, press Ctrl+C to exit")
        sys.stdin.read()

    except KeyboardInterrupt:
        print("\n[*] Exiting...")
    except Exception as e:
        print(f"[-] Error: {e}")

if __name__ == "__main__":
    run_frida_script("com.example.app", "hook_script.js")
```

5. 多设备管理

管理多台设备进行并行测试。

```
import subprocess
from concurrent.futures import ThreadPoolExecutor

def get_connected_devices():
    """获取所有已连接设备"""
    result = subprocess.run(["adb", "devices"], capture_output=True, text=True)
    lines = result.stdout.strip().split('\n')[1:]  # 跳过标题行
    devices = []
    for line in lines:
        if '\t' in line:
            device_id, status = line.split('\t')
            if status == 'device':
                devices.append(device_id)
    return devices

def run_on_device(device_id, task_func, *args):
    """在指定设备上运行任务"""
    adb = AdbWrapper(device_id)
    return task_func(adb, *args)

def run_on_all_devices(task_func, *args):
    """在所有设备上并行运行任务"""
    devices = get_connected_devices()
    print(f"[+] Found {len(devices)} devices")

    with ThreadPoolExecutor(max_workers=len(devices)) as executor:
        futures = [
            executor.submit(run_on_device, device_id, task_func, *args)
            for device_id in devices
        ]
        results = [f.result() for f in futures]
    return results

# 示例任务
def install_apk_task(adb, apk_path):
    return adb.install(apk_path)

def screenshot_task(adb, output_dir):
    device_id = adb.device_id or "default"
    local_path = f"{output_dir}/{device_id}_screenshot.png"
    adb.screenshot(local_path=local_path)
    return local_path

# 使用示例
if __name__ == "__main__":
    devices = get_connected_devices()
    print(f"Connected devices: {devices}")

    # 在所有设备上安装 APK
    # run_on_all_devices(install_apk_task, "./app.apk")
```

```
# 在所有设备上截图  
# run_on_all_devices(screenshot_task, "./screenshots")
```

总结

自动化脚本是 Android 逆向工程的重要工具。通过 Python 封装 ADB 命令、使用 UIAutomator2 进行 UI 自动化、结合 Frida 进行动态分析，可以构建强大的自动化测试和分析流程。

[R35] Native Hook 技术

R35: Native 层 Hook 技巧 (Native Hooking Patterns)

在 Android 逆向中，Native 层 (C/C++) 的分析往往比 Java 层更具挑战性。Hook 标准 C 库 (libc) 函数是理解 Native 层行为、脱壳和还原算法的重要手段。

1. 文件操作监控 (File I/O)

监控文件操作可以帮助我们发现 App 读取了哪些配置文件、加载了哪些 Dex/So 文件，或者将解密后的数据写入到了哪里。

Hook `open` / `openat`

```
// Hook open and openat to trace file access
function hookFileOpen() {
    // Intercept 'open'
    var openPtr = Module.findExportByName("libc.so", "open");
    if (openPtr) {
        Interceptor.attach(openPtr, {
            onEnter: function (args) {
                this.path = args[0].readCString();
                this.flags = args[1].toInt32();
            },
            onLeave: function (retval) {
                if (
                    this.path &&
                    (this.path.endsWith(".dex") || this.path.endsWith(".so"))
                ) {
                    console.log("[open] FD: " + retval + " Path: " + this.path);
                }
            },
        });
    }

    // Intercept 'openat' (commonly used on newer android versions)
    var openatPtr = Module.findExportByName("libc.so", "openat");
    if (openatPtr) {
        Interceptor.attach(openatPtr, {
            onEnter: function (args) {
                // args[0] is dirfd, args[1] is path
                this.path = args[1].readCString();
                this.flags = args[2].toInt32();
            },
            onLeave: function (retval) {
                if (
                    this.path &&
                    (this.path.indexOf("base.apk") >= 0 || this.path.indexOf(".dex") >= 0)
                ) {
                    console.log("[openat] FD: " + retval + " Path: " + this.path);
                }
            },
        });
    }
}

hookFileOpen();
```

2. 动态库加载监控 (dlopen)

监控 `dlopen` 可以帮助我们发现 App 动态加载了哪些 SO 文件，这对于分析加壳应用非常有用。

Hook `dlopen` / `android_dlopen_ext`

```
function hookDlopen() {
    var dlopen = Module.findExportByName(null, "dlopen");
    var android_dlopen_ext = Module.findExportByName(null, "android_dlopen_ext");

    if (dlopen) {
        Interceptor.attach(dlopen, {
            onEnter: function (args) {
                this.path = args[0].readCString();
            },
            onLeave: function (retval) {
                if (this.path) {
                    console.log("[dlopen] " + this.path + " -> Handle: " + retval);
                    if (this.path.indexOf("libnative-lib.so") >= 0) {
                        // Library loaded, ready to hook functions inside it
                        console.log("[+] Target library loaded!");
                    }
                }
            },
        });
    }

    if (android_dlopen_ext) {
        Interceptor.attach(android_dlopen_ext, {
            onEnter: function (args) {
                this.path = args[0].readCString();
            },
            onLeave: function (retval) {
                if (this.path) {
                    console.log(
                        "[android_dlopen_ext] " + this.path + " -> Handle: " + retval
                    );
                }
            },
        });
    }

    hookDlopen();
}
```

3. 内存操作监控 (memcpy)

监控 `memcpy` 可以帮助我们发现内存中的数据拷贝，特别是在脱壳时可以捕获解密后的 DEX 文件。

Hook memcpy

```
function hookMemcpy() {
    var memcpy = Module.findExportByName("libc.so", "memcpy");

    Interceptor.attach(memcpy, {
        onEnter: function (args) {
            this.dest = args[0];
            this.src = args[1];
            this.n = args[2].toInt32();
        },
        onLeave: function (retval) {
            // Filter by size or content to reduce noise
            if (this.n > 100 && this.n < 200) {
                // Check if source contains specific magic bytes (e.g., ELF header)
                try {
                    var magic = this.src.readU32();
                    if (magic == 0x464c457f) {
                        // .ELF
                        console.log("[memcpy] ELF header detected! Size: " + this.n);
                        console.log(hexdump(this.src, { length: 32 }));
                    }
                } catch (e) {}
            }

            // Check for DEX magic (dex\n035)
            if (this.n > 1000) {
                try {
                    var dexMagic = this.src.readUtf8String(4);
                    if (dexMagic === "dex\n") {
                        console.log("[memcpy] DEX file detected! Size: " + this.n);
                        // Dump DEX file
                        var dexData = this.src.readByteArray(this.n);
                        var filename = "/data/local/tmp/dump_" + Date.now() + ".dex";
                        var file = new File(filename, "wb");
                        file.write(dexData);
                        file.close();
                        console.log("[+] DEX dumped to: " + filename);
                    }
                } catch (e) {}
            }
        },
    });
}

hookMemcpy();
```

4. 符号解析监控 (dlsym)

监控 `dlsym` 可以帮助我们发现 App 动态查找了哪些函数，这对于分析混淆代码非常有用。

Hook `dlsym`

```
function hookDlsym() {
    var dlsym = Module.findExportByName(null, "dlsym");

    Interceptor.attach(dlsym, {
        onEnter: function (args) {
            this.handle = args[0];
            this.symbol = args[1].readCString();
        },
        onLeave: function (retval) {
            if (this.symbol) {
                console.log(
                    "[dlsym] Symbol: " + this.symbol + " -> Address: " + retval
                );
            }

            // Hook specific functions when resolved
            if (this.symbol === "encrypt" && !retval.isNull()) {
                console.log("[+] Found encrypt function, hooking...");
                hookNativeFunction(retval, "encrypt");
            }
        },
    });
}

function hookNativeFunction(addr, name) {
    Interceptor.attach(addr, {
        onEnter: function (args) {
            console.log("[" + name + "] called");
            console.log(" arg0: " + args[0]);
            console.log(" arg1: " + args[1]);
        },
        onLeave: function (retval) {
            console.log(" retval: " + retval);
        },
    });
}

hookDlsym();
```

5. 字符串比较监控 (strcmp)

监控字符串比较函数可以帮助我们发现 App 的校验逻辑，如 Root 检测、调试检测等。

Hook `strcmp` / `strstr`

```
function hookStringFunctions() {
    // Hook strcmp
    var strcmp = Module.findExportByName("libc.so", "strcmp");
    if (strcmp) {
        Interceptor.attach(strcmp, {
            onEnter: function (args) {
                var s1 = args[0].readCString();
                var s2 = args[1].readCString();

                // Filter for interesting strings
                var keywords = ["root", "su", "magisk", "frida", "xposed", "debug"];
                for (var i = 0; i < keywords.length; i++) {
                    if (
                        (s1 && s1.toLowerCase().indexOf(keywords[i]) >= 0) ||
                        (s2 && s2.toLowerCase().indexOf(keywords[i]) >= 0)
                    ) {
                        console.log("[strcmp] " + s1 + " VS " + s2);
                        break;
                    }
                }
            },
        });
    }

    // Hook strstr
    var strstr = Module.findExportByName("libc.so", "strstr");
    if (strstr) {
        Interceptor.attach(strstr, {
            onEnter: function (args) {
                this.haystack = args[0].readCString();
                this.needle = args[1].readCString();
            },
            onLeave: function (retval) {
                if (this.needle && this.needle.toLowerCase().indexOf("frida") >= 0) {
                    console.log(
                        "[strstr] Searching for: " +
                        this.needle +
                        " in: " +
                        this.haystack.substring(0, 50)
                    );
                    // Return NULL to bypass detection
                    retval.replace(ptr(0));
                }
            },
        });
    }

    hookStringFunctions();
}
```

6. 系统调用监控

监控系统调用可以帮助我们发现 App 的底层行为。

Hook `syscall`

```
function hookSyscall() {
    var syscall = Module.findExportByName("libc.so", "syscall");

    if (syscall) {
        Interceptor.attach(syscall, {
            onEnter: function (args) {
                var syscallNumber = args[0].toInt32();

                // Common syscall numbers on ARM64
                var syscallNames = {
                    56: "openat",
                    57: "close",
                    63: "read",
                    64: "write",
                    78: "readlinkat",
                    79: "fstatat",
                    101: "nanosleep",
                    172: "getpid",
                    174: "getuid",
                };

                if (syscallNames[syscallNumber]) {
                    console.log("[syscall] " + syscallNames[syscallNumber]);
                }
            },
        });
    }
}

hookSyscall();
```

7. 综合示例：脱壳辅助

结合多个 Hook 点进行脱壳分析。

```
// Comprehensive unpacking helper
function unpackHelper() {
    console.log("[+] Unpacking helper started");

    var dexCount = 0;

    // Hook mmap to catch memory mapping
    var mmap = Module.findExportByName("libc.so", "mmap");
    Interceptor.attach(mmap, {
        onEnter: function (args) {
            this.addr = args[0];
            this.length = args[1].toInt32();
            this.prot = args[2].toInt32();
        },
        onLeave: function (retval) {
            if (this.length > 100000 && this.prot == 5) {
                // PROT_READ | PROT_EXEC
                try {
                    var magic = retval.readUtf8String(4);
                    if (magic === "dex\n") {
                        console.log(
                            "[mmap] DEX detected! Size: " + this.length + " Address: " + retval
                        );
                        dexCount++;
                        var filename =
                            "/data/local/tmp/unpack_" + dexCount + "_" + Date.now() + ".dex";
                        var file = new File(filename, "wb");
                        file.write(retval.readByteArray(this.length));
                        file.close();
                        console.log("[+] Dumped to: " + filename);
                    }
                } catch (e) {}
            }
        },
    });
}

// Hook dvmDexFileOpenPartial (for Dalvik)
// Hook art::DexFile::Open (for ART)
var artDexOpen = Module.findExportByName(
    "libart.so",
    "_ZN3art7DexFile10openMemoryEPKhjRKNSt3__112basic_stringIcNS3_11char_traitsIcEENS3_9allocatorIcEEEEjPNS_6MemMapEPKNS_70atFileEPS9_"
);
if (artDexOpen) {
    Interceptor.attach(artDexOpen, {
        onEnter: function (args) {
            console.log("[art::DexFile::OpenMemory] called");
            this.base = args[0];
            this.size = args[1].toInt32();
        },
        onLeave: function (retval) {
```

```
        if (this.size > 0) {
            console.log("[+] ART DEX loaded, size: " + this.size);
        }
    },
});
}

unpackHelper();
```

总结

Native 层 Hook 是 Android 逆向的核心技能之一。通过 Hook libc 函数（如 open、dlopen、memcpy、strcmp 等），我们可以：

1. 监控文件操作，发现配置文件和动态加载的库
2. 跟踪内存操作，捕获解密后的数据
3. 分析字符串比较，绕过安全检测
4. 进行脱壳分析，提取被保护的 DEX 文件

在实践中，需要根据目标应用的具体行为来选择合适的 Hook 点。

[R36] Objection 代码片段

R36: Objection 常用技巧 (Objection Snippets)

Objection 是一个基于 Frida 开发的运行时移动端探索工具包。它提供了类似于 shell 的交互式命令行，无需编写 JavaScript 代码即可完成大部分常见的逆向任务。

- 安装: `pip install objection`
- 启动: `objection -g com.example.app explore`

1. 内存漫游与类查找

在不知道从何入手时，首先浏览应用中加载了哪些类。

- 搜索类:

```
# 搜索包含 "Crypto" 的类  
android hooking search classes Crypto
```

```
# 搜索包含 "encrypt" 的方法  
android hooking search methods encrypt
```

```
# 列出 com.example.app.MainActivity 的所有方法  
android hooking list class_methods com.example.app.MainActivity
```

2. Hook 方法

Objection 的核心功能之一是快速 Hook 类或方法，打印调用的参数、返回值和调用栈。

- Hook 整个类的所有方法:

```
android hooking watch class com.example.app.CryptoUtil
```

```
# 拦截 encrypt 方法，并打印参数和返回值  
android hooking watch class_method com.example.app.CryptoUtil.encrypt --dump-args --  
dump-return
```

```
# 强制 isRooted 方法返回 false  
android hooking set return_value com.example.app.Security.isRooted false
```

3. 堆操作

可以搜索内存中存在的对象实例，甚至调用这些实例的方法。

- 搜索堆中的实例：

```
# 查找内存中现存的 User 实例  
android heap search instances com.example.app.User
```

```
# 假设上一步搜索到实例 hashCode 为 123456  
# 调用该实例的 getToken 方法  
android heap execute 123456 getToken
```

```
# 查看该实例的 username 字段值  
android heap evaluate 123456  
# (进入编辑器后输入) console.log(clazz.username.value)
```

4. Activity 与 Fragment

- 查看当前 Activity：

```
android hooking get current_activity
```

```
android hooking list fragments
```

```
android intent launch_activity com.example.app.SecretActivity
```

5. 内存与 SO 库操作

- 列出加载的 SO 库:

```
memory list modules
```

```
# 将 libnative-lib.so 导出到本地文件（用于修复脱壳后的 SO）
memory dump from_base 0x7b12345000 0x50000 output.so
# 或自动下载
memory dump all libnative-lib.so
```

6. 文件系统操作

```
ls
cd cache
cat log.txt
file download /data/data/com.example.app/shared_prefs/config.xml
```

7. 安全绕过

- 禁用 SSL Pinning:

```
android sslpinning disable
```

```
android root disable
```

8. 导入自定义脚本

```
import /path/to/my_script.js
```

[R37] C 语言仿真脚本

R37: C 代码：用于运行时仿真与设备指纹生成

在逆向工程中，直接使用 C/C++ 编写一些辅助工具或重现目标逻辑是一种非常高效的策略。这可以帮助我们脱离复杂的 App 环境，对核心算法进行独立的测试、Fuzzing 或仿真。

1. 运行时仿真 (Runtime Emulation)

当我们在 SO 文件中定位到一个关键的核心算法（如自定义加密、签名生成）后，如果该算法逻辑清晰且依赖较少，最好的方法就是将其逻辑用 C/C++ "翻译"一遍。

场景示例：重现一个简单的 XOR 加密算法

假设在 IDA Pro 中看到如下伪代码：

```
// Decompiled pseudo-code from IDA
void encrypt_data(char* data, int len) {
    for (int i = 0; i < len; ++i) {
        data[i] = (data[i] ^ 0x5A) + 5;
    }
}
```

重现与验证代码

```
// emulate_encrypt.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>

// Re-implementation of the encryption algorithm
void simulate_encrypt(char* data, size_t len) {
    for (size_t i = 0; i < len; ++i) {
        data[i] = (data[i] ^ 0x5A) + 5;
    }
}

// Corresponding decryption for our own testing
void simulate_decrypt(char* data, size_t len) {
    for (size_t i = 0; i < len; ++i) {
        data[i] = (data[i] - 5) ^ 0x5A;
    }
}

// Helper function: Print hexadecimal
void print_hex(const char* label, uint8_t* data, size_t len) {
    printf("%s: ", label);
    for (size_t i = 0; i < len; i++) {
        printf("%02x ", data[i]);
    }
    printf("\n");
}

int main() {
    char my_data[] = "this_is_a_test_message";
    size_t len = strlen(my_data);

    printf("==> XOR Encryption Algorithm Test ==>\n\n");
    printf("Original: %s\n", my_data);
    print_hex("Original HEX", (uint8_t*)my_data, len);
    printf("\n");

    // Encrypt it
    simulate_encrypt(my_data, len);
    printf("After encryption:\n");
    print_hex("Encrypted HEX", (uint8_t*)my_data, len);
    printf("\n");

    // Decrypt it
    simulate_decrypt(my_data, len);
    printf("After decryption: %s\n", my_data);
    print_hex("Decrypted HEX", (uint8_t*)my_data, len);
    printf("\n");

    return 0;
}
```

编译与运行

```
# 编译  
gcc emulate_encrypt.c -o emulate  
  
# 运行  
../emulate  
  
# 输出:  
# === XOR Encryption Algorithm Test ===  
#  
# Original: this_is_a_test_message  
# Original HEX: 74 68 69 73 5f 69 73 5f 61 5f 74 65 73 74 5f 6d ...  
#  
# After encryption:  
# Encrypted HEX: 29 37 38 2e 0a 38 2e 0a 3e 0a 29 3a 2e 29 0a 32 ...  
#  
# After decryption: this_is_a_test_message
```

2. 设备指纹生成 (Device Fingerprint Generation)

许多 App 会通过读取 Android 系统的 `build.prop` 或其他系统属性来生成设备指纹，用于识别和跟踪设备。在进行自动化操作时，我们需要能够模拟这些指纹。

`getprop` 是 Android shell 中的一个命令，可以读取系统属性。我们也可以用 C 代码在 Native 层实现类似的功能，从而生成可以乱真的指纹数据。

场景示例：用 C 读取关键设备属性并生成 JSON

```
// device_fingerprint.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// A simple wrapper to execute a shell command and get its output
// In a real scenario, you might use direct system calls for better performance/
stealth
char* get_prop(const char* key) {
    char command[256];
    snprintf(command, sizeof(command), "getprop %s", key);

    FILE* fp = popen(command, "r");
    if (fp == NULL) {
        return NULL;
    }

    char* line = malloc(256);
    if (fgets(line, 256, fp) == NULL) {
        free(line);
        pclose(fp);
        return NULL;
    }

    // Remove trailing newline
    line[strcspn(line, "\n")] = 0;
    pclose(fp);
    return line;
}

int main() {
    // List of properties we want to fetch
    const char* props_to_fetch[] = {
        "ro.product.brand",
        "ro.product.model",
        "ro.product.manufacturer",
        "ro.product.device",
        "ro.build.version.release",
        "ro.build.version.sdk",
        "ro.build.fingerprint",
        "ro.serialno",
        "ro.boot.serialno"
    };
    int num_props = sizeof(props_to_fetch) / sizeof(props_to_fetch[0]);

    printf("{\n");
    printf("  \"timestamp\": %ld,\n", time(NULL));
    printf("  \"device\": {\n");

    for (int i = 0; i < num_props; ++i) {
        char* value = get_prop(props_to_fetch[i]);
```

```
if (value) {
    // Extract last part of property name for cleaner key
    const char* last_dot = strrchr(props_to_fetch[i], '.');
    const char* key = last_dot ? last_dot + 1 : props_to_fetch[i];

    printf("    \"%s\": \"%s\"", key, value);
    if (i < num_props - 1) {
        printf(",");
    }
    printf("\n");
    free(value);
}
}

printf(" }\n");
printf("}\n");

return 0;
}
```

编译与运行

使用 Android NDK 进行交叉编译:

```
# 设置NDK路径（根据实际安装位置修改）
export NDK_PATH=~/Android/Sdk/ndk/25.1.8937393

# 编译（arm64 架构）
$NDK_PATH/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android21-clang \
device_fingerprint.c -o fingerprint

# 推送到设备
adb push fingerprint /data/local/tmp/
adb shell chmod +x /data/local/tmp/fingerprint

# 运行
adb shell /data/local/tmp/fingerprint
```

输出示例

```
{  
    "timestamp": 1734518400,  
    "device": {  
        "brand": "google",  
        "model": "Pixel 5",  
        "manufacturer": "Google",  
        "device": "redfin",  
        "release": "13",  
        "sdk": "33",  
        "fingerprint": "google/redfin/redfin:13/TQ3A.230901.001/10750268:user/release-  
keys",  
        "serialno": "XXXXXXXXXX"  
    }  
}
```

3. 更复杂的算法仿真

示例：HMAC-SHA256 签名仿真

```
// hmac_sign.c
#include <stdio.h>
#include <string.h>
#include <openssl/hmac.h>
#include <openssl/sha.h>

void hmac_sha256(const char* key, const char* data, unsigned char* result) {
    unsigned int len = 32;
    HMAC(EVP_sha256(),
          key, strlen(key),
          (unsigned char*)data, strlen(data),
          result, &len);
}

void print_hex(unsigned char* data, size_t len) {
    for (size_t i = 0; i < len; i++) {
        printf("%02x", data[i]);
    }
    printf("\n");
}

int main() {
    const char* key = "secret_key_12345";
    const char* data = "user=test&timestamp=1234567890";

    unsigned char result[32];
    hmac_sha256(key, data, result);

    printf("HMAC-SHA256 Signature:\n");
    print_hex(result, 32);

    return 0;
}
```

编译

```
# 需要链接 OpenSSL
gcc hmac_sign.c -o hmac_sign -lssl -lcrypto

# 运行
./hmac_sign
```

4. 使用 Unidbg 进行 SO 仿真

对于复杂的 Native 算法，可以使用 Unidbg 进行完整仿真，而不需要手动翻译代码。

```
// Java 代码示例 (Unidbg)
import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.VM;

public class EmulatorExample {
    private AndroidEmulator emulator;
    private VM vm;
    private DalvikModule dm;

    public EmulatorExample() {
        // 创建模拟器
        emulator = AndroidEmulatorBuilder
            .for64Bit()
            .setProcessName("com.example.app")
            .build();

        // 创建虚拟机
        vm = emulator.createDalvikVM();

        // 加载 SO 文件
        dm = vm.loadLibrary("libnative-lib.so", false);

        // 调用 JNI_OnLoad
        dm.callJNI_OnLoad(emulator);
    }

    public String callEncrypt(String input) {
        // 调用 Native 方法
        // ... (具体实现取决于目标函数签名)
        return "";
    }
}
```

总结

使用 C/C++ 进行运行时仿真的优势：

1. 独立测试：脱离 App 环境，可以快速迭代测试
2. 便于 Fuzzing：可以对算法进行大规模模糊测试
3. 性能优越：相比在 App 内 Hook，独立运行效率更高
4. 便于分享：可以将仿真代码分享给团队其他成员

在实践中，建议结合动态分析（Frida Hook）和静态分析（IDA Pro）的结果，逐步完善仿真代码，直到输出与目标函数完全一致。

Native 开发

[R38] JNI 开发与调用原理

R38: JNI 开发与调用原理

JNI (Java Native Interface) 是 Java 与 C/C++ 代码交互的标准接口。在 Android 逆向工程中，理解 JNI 的工作原理对于分析 Native 层代码至关重要。本章将从开发者视角介绍 JNI 的原理与实践。

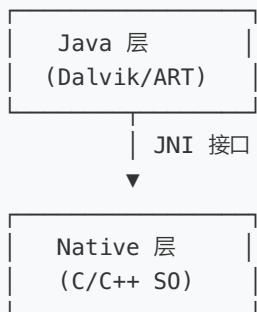
目录

- [1. JNI 基础概念](#)
- [2. JNI 开发环境搭建](#)
- [3. JNI 函数注册方式](#)
- [4. JNI 数据类型映射](#)
- [5. JNI 常用函数](#)
- [6. 完整示例：实现加密函数](#)
- [7. 逆向分析要点](#)

JNI 基础概念

什么是 JNI

JNI 是一个编程框架，允许 Java 代码调用 Native 代码（C/C++），也允许 Native 代码调用 Java 代码。



为什么使用 JNI

用途	说明
性能优化	计算密集型任务（加密、图像处理）使用 C/C++ 更高效
代码复用	复用现有的 C/C++ 库（OpenSSL、FFmpeg 等）
安全保护	Native 代码比 Java 更难逆向，常用于核心算法保护
硬件访问	直接访问硬件或操作系统底层功能

JNI 开发环境搭建

1. 配置 NDK

在 `build.gradle` 中配置 NDK:

```
android {  
    defaultConfig {  
        ndk {  
            abiFilters 'arm64-v8a', 'armeabi-v7a'  
        }  
    }  
  
    externalNativeBuild {  
        cmake {  
            path "src/main/cpp/CMakeLists.txt"  
        }  
    }  
}
```

2. 创建 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10.2)  
  
project("nativelib")  
  
add_library(  
    native-lib  
    SHARED  
    native-lib.cpp  
)  
  
find_library(  
    log-lib  
    log  
)  
  
target_link_libraries(  
    native-lib  
    ${log-lib}  
)
```

3. 目录结构

```
app/
└── src/main/
    ├── java/com/example/app/
    │   └── NativeHelper.java      # Java 层声明
    └── cpp/
        ├── CMakeLists.txt
        └── native-lib.cpp         # C/C++ 实现
```

JNI 函数注册方式

方式一：静态注册（自动关联）

Java 层声明 native 方法：

```
public class NativeHelper {
    static {
        System.loadLibrary("native-lib");
    }

    // 声明 native 方法
    public native String encrypt(String input);
    public native byte[] decrypt(byte[] data);
}
```

C++ 层实现（函数名遵循特定规则）：

```
#include <jni.h>
#include <string>

// 函数名格式: Java_包名_类名_方法名
// 包名中的 . 替换为 _
extern "C" JNIEXPORT jstring JNICALL
Java_com_example_app_NativeHelper_encrypt(
    JNIEnv *env,
    jobject thiz,      // 非静态方法: jobject 指向调用对象
    jstring input) {

    // 获取 Java 字符串
    const char *str = env->GetStringUTFChars(input, nullptr);

    // 处理逻辑...
    std::string result = "encrypted_" + std::string(str);

    // 释放资源
    env->ReleaseStringUTFChars(input, str);

    // 返回新的 Java 字符串
    return env->NewStringUTF(result.c_str());
}
```

静态注册的函数命名规则：

Java_<包名>_<类名>_<方法名>

示例：

- 包名: com.example.app
- 类名: NativeHelper
- 方法名: encrypt
- 函数名: Java_com_example_app_NativeHelper_encrypt

方式二：动态注册 (JNI_OnLoad)

动态注册在 `JNI_OnLoad` 中手动绑定函数：

```
#include <jni.h>
#include <string>

// Native 函数实现 (函数名可以任意)
static jstring native_encrypt(JNIEnv *env, jobject thiz, jstring input) {
    const char *str = env->GetStringUTFChars(input, nullptr);
    std::string result = "encrypted_" + std::string(str);
    env->ReleaseStringUTFChars(input, str);
    return env->NewStringUTF(result.c_str());
}

static jbyteArray native_decrypt(JNIEnv *env, jobject thiz, jbyteArray data) {
    // 解密实现...
    return data;
}

// 方法映射表
static(JNIEnv gMethods[] = {
    // {Java方法名, 方法签名, Native函数指针}
    {"encrypt", "(Ljava/lang/String;)Ljava/lang/String;", (void*)native_encrypt},
    {"decrypt", "([B)[B", (void*)native_decrypt},
};

// JNI_OnLoad: 库加载时自动调用
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved) {
    JNIEnv *env = nullptr;

    if (vm->GetEnv((void**)&env, JNI_VERSION_1_6) != JNI_OK) {
        return JNI_ERR;
    }

    // 找到目标 Java 类
    jclass clazz = env->FindClass("com/example/app/NativeHelper");
    if (clazz == nullptr) {
        return JNI_ERR;
    }

    // 注册Native 方法
    int methodCount = sizeof(gMethods) / sizeof(gMethods[0]);
    if (env->RegisterNatives(clazz, gMethods, methodCount) < 0) {
        return JNI_ERR;
    }

    return JNI_VERSION_1_6;
}
```

两种注册方式对比

特性	静态注册	动态注册
函数名	必须遵循命名规则	可以任意命名
查找时机	首次调用时查找	加载时一次性注册
性能	略慢 (需要查找)	较快 (直接使用函数指针)
安全性	函数名暴露类/方法信息	函数名可混淆, 更隐蔽
逆向难度	较容易定位	需要分析 JNI_OnLoad

JNI 数据类型映射

基本类型

Java 类型	JNI 类型	签名	C/C++ 类型
boolean	jboolean	Z	uint8_t
byte	jbyte	B	int8_t
char	jchar	C	uint16_t
short	jshort	S	int16_t
int	jint	I	int32_t
long	jlong	J	int64_t
float	jfloat	F	float
double	jdouble	D	double
void	void	V	void

引用类型

Java 类型	JNI 类型	签名
Object	jobject	Ljava/lang/Object;
String	jstring	Ljava/lang/String;
Class	jclass	Ljava/lang/Class;
int[]	jintArray	[I
byte[]	jByteArray	[B
Object[]	jobjectArray	[Ljava/lang/Object;

方法签名格式

(参数类型) 返回类型

示例:

```
- void method()           -> ()V
- int method(int a)       -> (I)I
- String method(String s, int i) -> (Ljava/lang/String;I)Ljava/lang/String;
- byte[] method(byte[] data) -> ([B][B
- void method(int[] arr, Object o) -> ([ILjava/lang/Object;)V
```

JNI 常用函数

字符串操作

```
// Java String -> C 字符串  
const char* str = env->GetStringUTFChars(javaString, nullptr);  
// 使用完毕后释放  
env->ReleaseStringUTFChars(javaString, str);  
  
// C 字符串 -> Java String  
jstring result = env->NewStringUTF("Hello from C++");  
  
// 获取字符串长度  
jsize len = env->GetStringUTFLength(javaString);
```

数组操作

```
// 获取数组长度  
jsize len = env->GetArrayLength(javaArray);  
  
// byte[] 操作  
jbyte* bytes = env->GetByteArrayElements(byteArray, nullptr);  
// 使用完毕后释放  
env->ReleaseByteArrayElements(byteArray, bytes, 0);  
  
// 创建新的 byte[]  
jbyteArray newArray = env->NewByteArray(length);  
env->SetByteArrayRegion(newArray, 0, length, data);  
  
// int[] 操作  
jint* ints = env->GetIntArrayElements(intArray, nullptr);  
env->ReleaseIntArrayElements(intArray, ints, 0);
```

调用 Java 方法

```
// 1. 获取类引用  
jclass clazz = env->FindClass("com/example/app/MyClass");  
  
// 2. 获取方法 ID  
jmethodID methodId = env->GetMethodID(clazz, "methodName", "(I)Ljava/lang/String;");  
// 静态方法使用 GetStaticMethodID  
  
// 3. 调用方法  
jstring result = (jstring)env->CallObjectMethod(obj, methodId, 123);  
// 静态方法使用 CallStaticObjectMethod
```

访问 Java 字段

```
// 获取字段 ID  
jfieldID fieldId = env->GetFieldID(clazz, "fieldName", "I");  
// 静态字段使用 GetStaticFieldID  
  
// 读取字段值  
jint value = env->GetIntField(obj, fieldId);  
  
// 设置字段值  
env->SetIntField(obj, fieldId, 100);
```

完整示例：实现加密函数

Java 层

```
package com.example.crypto;

public class CryptoHelper {
    static {
        System.loadLibrary("crypto-lib");
    }

    // AES 加密
    public native byte[] aesEncrypt(byte[] data, byte[] key);

    // AES 解密
    public native byte[] aesDecrypt(byte[] data, byte[] key);

    // MD5 哈希
    public native String md5(String input);
}
```

Native 层 (crypto-lib.cpp)

```
#include <jni.h>
#include <string>
#include <cstring>
#include <android/log.h>

#define LOG_TAG "CryptoLib"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)

// 简化的 XOR 加密示例 (实际应使用 OpenSSL 等库)
static jbyteArray xor_encrypt(JNIEnv *env, jbyteArray data, jbyteArray key) {
    jsize dataLen = env->GetArrayLength(data);
    jsize keyLen = env->GetArrayLength(key);

    jbyte* dataBytes = env->GetByteArrayElements(data, nullptr);
    jbyte* keyBytes = env->GetByteArrayElements(key, nullptr);

    // 创建结果数组
    jbyteArray result = env->NewByteArray(dataLen);
    jbyte* resultBytes = new jbyte[dataLen];

    // XOR 加密
    for (int i = 0; i < dataLen; i++) {
        resultBytes[i] = dataBytes[i] ^ keyBytes[i % keyLen];
    }

    env->SetByteArrayRegion(result, 0, dataLen, resultBytes);

    // 清理
    delete[] resultBytes;
    env->ReleaseByteArrayElements(data, dataBytes, 0);
    env->ReleaseByteArrayElements(key, keyBytes, 0);

    return result;
}

// 简化的 MD5 实现 (示例)
static jstring calc_md5(JNIEnv *env, jstring input) {
    const char* str = env->GetStringUTFChars(input, nullptr);

    // 实际应使用 OpenSSL 的 MD5 函数
    // 这里仅作示例, 返回模拟的 MD5
    char md5[33];
    sprintf(md5, sizeof(md5), "%032x", (unsigned int)strlen(str));

    env->ReleaseStringUTFChars(input, str);
    return env->NewStringUTF(md5);
}

// 方法映射表
static JNINativeMethod gMethods[] = {
    {"aesEncrypt", "(Ljava/nio/ByteBuffer;Ljava/nio/ByteBuffer;)V", (void*)xor_encrypt},
    {"aesDecrypt", "(Ljava/nio/ByteBuffer;Ljava/nio/ByteBuffer;)V", // XOR 加解密相同
}
```

```
{"md5", "(Ljava/lang/String;)Ljava/lang/String;", (void*)calc_md5},  
};  
  
JNIEXPORT jint JNI_OnLoad(JavaVM *vm, void *reserved) {  
    JNIEnv *env = nullptr;  
  
    if (vm->GetEnv((void**)&env, JNI_VERSION_1_6) != JNI_OK) {  
        return JNI_ERR;  
    }  
  
    jclass clazz = env->FindClass("com/example/crypto/CryptoHelper");  
    if (clazz == nullptr) {  
        LOGI("Failed to find class");  
        return JNI_ERR;  
    }  
  
    int methodCount = sizeof(gMethods) / sizeof(gMethods[0]);  
    if (env->RegisterNatives(clazz, gMethods, methodCount) < 0) {  
        LOGI("Failed to register natives");  
        return JNI_ERR;  
    }  
  
    LOGI("JNI_OnLoad success, registered %d methods", methodCount);  
    return JNI_VERSION_1_6;  
}
```

逆向分析要点

1. 定位 JNI 函数

静态注册：

```
# 在 IDA/Ghidra 中搜索函数名  
# 格式: Java_包名_类名_方法名  
strings libnative.so | grep "Java_"
```

动态注册：

```
# 分析 JNI_OnLoad 函数  
# 查找 RegisterNatives 调用  
# 追踪 JNINativeMethod 数组
```

2. Frida Hook JNI 函数

```
// Hook JNI_OnLoad
Interceptor.attach(Module.findExportByName("libnative.so", "JNI_OnLoad"), {
    onEnter: function (args) {
        console.log("JNI_OnLoad called");
    },
    onLeave: function (retval) {
        console.log("JNI_OnLoad returned:", retval);
    },
});

// Hook RegisterNatives 获取动态注册信息
var RegisterNatives = Module.findExportByName(
    "libart.so",
    "_ZN3art3JNI15RegisterNativesEP7_JNIEnvP7_jclassPK15JNINativeMethodi"
);
Interceptor.attach(RegisterNatives, {
    onEnter: function (args) {
        var className = Java.vm.getEnv().getClassName(args[1]);
        var methods = args[2];
        var methodCount = args[3].toInt32();

        console.log("RegisterNatives:", className, "count:", methodCount);

        for (var i = 0; i < methodCount; i++) {
            var methodName = methods
                .add(i * Process.pointerSize * 3)
                .readPointer()
                .readCString();
            var signature = methods
                .add(i * Process.pointerSize * 3 + Process.pointerSize)
                .readPointer()
                .readCString();
            var fnPtr = methods
                .add(i * Process.pointerSize * 3 + Process.pointerSize * 2)
                .readPointer();

            console.log(" Method:", methodName, signature, "->", fnPtr);
        }
    },
});
```

3. 常见保护手段

保护手段	说明	应对方法
函数名混淆	使用动态注册隐藏函数名	分析 JNI_OnLoad
字符串加密	加密关键字符串	动态调试获取解密后的值
反调试检测	检测调试器/Frida	绕过反调试
代码混淆	OLLLVM 等混淆	符号执行/模式识别

[R39] SO 动态库编译与加载

R39: SO 动态库编译与加载

本章介绍如何从零开始编译 Android SO 动态库，以及 Android 系统加载 SO 的完整流程。理解这些原理对于逆向分析 Native 层代码至关重要。

目录

1. SO 文件基础
2. 编译环境准备
3. 使用 NDK 编译 SO
4. 使用 CMake 构建
5. SO 加载流程分析
6. 手动加载 SO
7. 调试与分析技巧

SO 文件基础

什么是 SO 文件

SO (Shared Object) 是 Linux/Android 系统的动态链接库格式，相当于 Windows 的 DLL。它采用 ELF (Executable and Linkable Format) 格式。

SO 文件结构

ELF Header	文件类型、架构、入口点
Program Headers	段加载信息
.text	可执行代码
.rodata	只读数据（字符串常量等）
.data	已初始化的全局变量
.bss	未初始化的全局变量
.dynamic	动态链接信息
.dynsym	动态符号表
.dynstr	动态字符串表
Section Headers	节信息

Android 支持的 CPU 架构

ABI	架构	说明
arm64-v8a	ARMv8-A (64 位)	现代 Android 设备主流
armeabi-v7a	ARMv7-A (32 位)	老旧设备兼容
x86_64	x86-64 (64 位)	模拟器、部分平板
x86	x86 (32 位)	老旧模拟器

编译环境准备

1. 安装 Android NDK

```
# 方式一: 通过 Android Studio SDK Manager 安装

# 方式二: 命令行下载
# 下载地址: https://developer.android.com/ndk/downloads
wget https://dl.google.com/android/repository/android-ndk-r25c-linux.zip
unzip android-ndk-r25c-linux.zip
export NDK_HOME=/path/to/android-ndk-r25c
export PATH=$PATH:$NDK_HOME
```

2. 验证安装

```
# 检查 NDK 版本
$NDK_HOME/ndk-build --version

# 检查可用的工具链
ls $NDK_HOME/toolchains/llvm/prebuilt/*/bin/
```

3. 独立工具链（可选）

```
# 创建独立工具链 (适合命令行编译)
$NDK_HOME/build/tools/make_standalone_toolchain.py \
    --arch arm64 \
    --api 21 \
    --install-dir /path/to/toolchain
```

使用 NDK 编译 SO

方式一：ndk-build

1. 项目结构

```
project/
└── jni/
    ├── Android.mk
    ├── Application.mk
    └── native.c
└── libs/
    ├── arm64-v8a/
    │   └── libnative.so
    └── armeabi-v7a/
        └── libnative.so
```

2. Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

# 模块名称 (生成 lib<名称>.so)
LOCAL_MODULE := native

# 源文件
LOCAL_SRC_FILES := native.c

# 依赖的系统库
LOCAL_LDLIBS := -llog -lz

# 编译选项
LOCAL_CFLAGS := -Wall -O2

# 构建动态库
include $(BUILD_SHARED_LIBRARY)
```

3. Application.mk

```
# 目标 ABI  
APP_ABI := arm64-v8a armeabi-v7a  
  
# 最低 API 级别  
APP_PLATFORM := android-21  
  
# C++ STL 库  
APP_STL := c++_shared  
  
# 优化级别  
APP_OPTIM := release
```

4. native.c

```
#include <jni.h>  
#include <string.h>  
#include <android/log.h>  
  
#define LOG_TAG "NativeLib"  
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)  
  
// 导出函数  
JNIEXPORT jstring JNICALL Java_com_example_app_MainActivity_stringFromJNI(JNIEnv *env, jobject thiz) {  
    LOGI("stringFromJNI called");  
    return (*env)->NewStringUTF(env, "Hello from C!");  
}  
  
// 初始化函数 (可选)  
_attribute__((constructor))  
void lib_init() {  
    LOGI("Library loaded!");  
}  
  
// 卸载函数 (可选)  
_attribute__((destructor))  
void lib_fini() {  
    LOGI("Library unloaded!");  
}
```

5. 编译

```
cd project/  
$NDK_HOME/ndk-build  
  
# 清理  
$NDK_HOME/ndk-build clean  
  
# 指定 verbose 输出  
$NDK_HOME/ndk-build V=1
```

方式二：命令行直接编译

```
# 设置编译器  
export CC=$NDK_HOME/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android21-  
clang  
  
# 编译为目标文件  
$CC -c -fPIC -o native.o native.c  
  
# 链接为 SO  
$CC -shared -o libnative.so native.o -llog  
  
# 查看导出符号  
$NDK_HOME/toolchains/llvm/prebuilt/linux-x86_64/bin/llvm-nm -D libnative.so
```

使用 CMake 构建

1. CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10.2)

project("nativelib")

# 设置 C++ 标准
set(CMAKE_CXX_STANDARD 17)

# 添加编译选项
add_compile_options(-Wall -O2)

# 定义库
add_library(
    native-lib
    SHARED
    src/native.cpp
    src/crypto.cpp
    src/utils.cpp
)

# 查找系统库
find_library(log-lib log)
find_library(z-lib z)

# 包含头文件目录
target_include_directories(native-lib PRIVATE
    ${CMAKE_SOURCE_DIR}/include
)

# 链接库
target_link_libraries(native-lib
    ${log-lib}
    ${z-lib}
)

# 添加预处理器定义
target_compile_definitions(native-lib PRIVATE
    DEBUG_MODE=1
)
```

2. 使用 CMake 编译

```
# 创建构建目录  
mkdir build && cd build  
  
# 配置 (指定 NDK 工具链)  
cmake \  
    -DCMAKE_TOOLCHAIN_FILE=$NDK_HOME/build/cmake/android.toolchain.cmake \  
    -DANDROID_ABI=arm64-v8a \  
    -DANDROID_PLATFORM=android-21 \  
    ..  
  
# 编译  
cmake --build .  
  
# 或者使用 make  
make -j4
```

3. Gradle 集成

```
android {  
    defaultConfig {  
        externalNativeBuild {  
            cmake {  
                cppFlags "-std=c++17 -frtti -fexceptions"  
                arguments "-DANDROID_STL=c++_shared"  
            }  
        }  
        ndk {  
            abiFilters 'arm64-v8a', 'armeabi-v7a'  
        }  
    }  
  
    externalNativeBuild {  
        cmake {  
            path "src/main/cpp/CMakeLists.txt"  
            version "3.10.2"  
        }  
    }  
}
```

SO 加载流程分析

Android SO 加载流程

```
Java: System.loadLibrary("native")
      |
      ▼
Runtime.loadLibrary0()
      |
      ▼
ClassLoader.findLibrary() —> 查找 libnative.so 路径
      |
      ▼
Runtime.nativeLoad()
      |
      ▼
dlopen("libnative.so") —> 系统加载器
      |
      └── 解析 ELF 头
      └── 映射 PT_LOAD 段到内存
      └── 处理依赖库 (递归加载)
      └── 执行重定位
      └── 调用 .init_array / constructor
      |
      ▼
JNI_OnLoad() —> 如果存在, 执行初始化
```

关键函数

函数	说明
<code>System.loadLibrary()</code>	Java 层加载库
<code>dlopen()</code>	Native 层打开动态库
<code>dlsym()</code>	查找符号地址
<code>dlclose()</code>	关闭动态库
<code>JNI_OnLoad()</code>	JNI 初始化回调
<code>JNI_OnUnload()</code>	JNI 卸载回调

库搜索路径

```
// 应用私有库目录  
/data/app/<package>/lib/<abi>/  
  
// 系统库目录  
/system/lib64/ (64位)  
/system/lib/ (32位)  
/vendor/lib64/  
/vendor/lib/
```

手动加载 SO

Java 层加载

```
public class NativeLoader {  
  
    // 方式一: 从默认路径加载  
    static {  
        System.loadLibrary("native"); // 加载 libnative.so  
    }  
  
    // 方式二: 从绝对路径加载  
    public static void loadFromPath(String path) {  
        System.load(path); // 如: /data/local/tmp/libnative.so  
    }  
  
    // 方式三: 自定义 ClassLoader  
    public static void loadWithClassLoader(String libName) {  
        try {  
            // 获取应用的 ClassLoader  
            ClassLoader loader = NativeLoader.class.getClassLoader();  
  
            // 使用反射调用 findLibrary  
            Method findLibrary = ClassLoader.class.getDeclaredMethod(  
                "findLibrary", String.class);  
            findLibrary.setAccessible(true);  
  
            String libPath = (String) findLibrary.invoke(loader, libName);  
            if (libPath != null) {  
                System.load(libPath);  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Native 层加载

```
#include <dlfcn.h>
#include <android/log.h>

#define LOG_TAG "DynamicLoader"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)

// 动态加载 SO 并调用函数
void load_and_call(const char* lib_path, const char* func_name) {
    // 打开动态库
    void* handle = dlopen(lib_path, RTLD_NOW);
    if (!handle) {
        LOGI("dlopen failed: %s", dlerror());
        return;
    }

    // 查找函数符号
    typedef int (*func_ptr)(int, int);
    func_ptr func = (func_ptr)dlsym(handle, func_name);

    if (!func) {
        LOGI("dlsym failed: %s", dlerror());
        dlclose(handle);
        return;
    }

    // 调用函数
    int result = func(10, 20);
    LOGI("Function result: %d", result);

    // 关闭库
    dlclose(handle);
}

// 示例: 加载加密库
void load_crypto_lib() {
    void* handle = dlopen("libcrypto.so", RTLD_LAZY);
    if (handle) {
        // 获取加密函数
        void* encrypt_func = dlsym(handle, "AES_encrypt");
        if (encrypt_func) {
            LOGI("Found AES_encrypt at %p", encrypt_func);
        }
    }
}
```

调试与分析技巧

1. 查看 SO 信息

```
# 查看 ELF 头信息  
readelf -h libnative.so  
  
# 查看程序头 (加载段)  
readelf -l libnative.so  
  
# 查看节头  
readelf -S libnative.so  
  
# 查看动态符号表  
readelf -s libnative.so  
  
# 查看依赖库  
readelf -d libnative.so | grep NEEDED  
  
# 查看导出函数  
nm -D libnative.so | grep " T "
```

2. Frida 动态分析

```
// 监控 dlopen
Interceptor.attach(Module.findExportByName(null, "dlopen"), {
    onEnter: function (args) {
        this.path = args[0].readCString();
        console.log("[dlopen]", this.path);
    },
    onLeave: function (retval) {
        console.log("[dlopen] handle:", retval);
    },
});

// 监控 dlsym
Interceptor.attach(Module.findExportByName(null, "dlsym"), {
    onEnter: function (args) {
        this.symbol = args[1].readCString();
        console.log("[dlsym]", this.symbol);
    },
    onLeave: function (retval) {
        console.log("[dlsym] address:", retval);
    },
});

// 枚举已加载的模块
Process.enumerateModules().forEach(function (module) {
    if (module.name.indexOf("native") !== -1) {
        console.log(module.name, module.base, module.size);
    }
});

// 枚举模块导出函数
Module.enumerateExports("libnative.so").forEach(function (exp) {
    console.log(exp.type, exp.name, exp.address);
});
```

3. GDB 调试

```
# 连接到进程  
adb forward tcp:1234 tcp:1234  
gdbserver :1234 --attach <pid>  
  
# 本地 GDB 连接  
$NDK_HOME/prebuilt/linux-x86_64/bin/gdb  
(gdb) target remote :1234  
(gdb) set solib-search-path /path/to/symbols  
  
# 在函数上设置断点  
(gdb) break Java_com_example_app_MainActivity_stringFromJNI  
(gdb) continue  
  
# 查看内存  
(gdb) x/20x $pc
```

4. 常见问题排查

问题	原因	解决方法
UnsatisfiedLinkError	找不到库或符号	检查库路径和函数签名
dlopen failed: library not found	依赖库缺失	检查 readelf -d 输出
text relocations	代码段重定位	编译时添加 -fPIC
cannot locate symbol	符号未导出	检查函数是否标记为导出

工具指南

动态分析工具

[T01] Frida 使用指南

T01: Frida 常用命令与脚本 API 大全

Frida 是一个动态代码插桩工具包，它允许您将自己的脚本注入到黑盒进程中。它对于逆向工程、安全研究和应用调试非常有用。

目录

- [Frida 工具集](#)
 - [连接与附加模式](#)
 - [JavaScript API \(核心\)](#)
 - [Java \(Android\)](#)
 - [Objective-C \(iOS\)](#)
 - [通用/原生 \(Native\)](#)
 - [JNI \(Java Native Interface\)](#)
 - [常用脚本示例](#)
-

Frida 工具集

这些是在终端中使用的核心 Frida 命令行工具。

命令	描述
<code>frida --version</code>	查看 Frida 版本
<code>frida-ps -U</code>	列出 USB 连接设备上的所有进程
<code>frida-ps -Ua</code>	列出 USB 连接设备上所有正在运行的应用程序
<code>frida-ps -Uai</code>	列出 USB 连接设备上所有已安装的应用程序及其标识符
<code>frida-trace -U -f <包名> -i "<方法>"</code>	跟踪指定方法的调用（附加到新进程）
<code>frida-trace -U -p <PID> -i "<方法>"</code>	跟踪指定方法的调用（附加到现有进程）
<code>frida -U -f <包名> -l <脚本.js></code>	Spawn 一个新进程并注入脚本
<code>frida -U -p <PID> -l <脚本.js></code>	附加到现有进程并注入脚本
<code>frida -U --no-pause -f <包名> -l <脚本.js></code>	Spawn 新进程并注入脚本，且不暂停主线程

连接与附加模式

Frida 有两种主要的方式来 hook 应用：

- **Spawn (Spawning):** Frida 启动应用程序并立即暂停主线程，以便您在应用代码执行前注入脚本。这是最常用的模式，特别是对于需要在应用启动早期进行 Hook 的场景。使用 `-f <包名>` 参数。
- **Attach (Attaching):** Frida 附加到已经在运行的进程上。这对于 hook 那些在应用运行中途才会触发的功能很有用。使用 `-p <PID>` 或应用名称。

JavaScript API (核心)

这是 Frida 脚本的核心。所有逻辑都在 JavaScript 脚本中实现。

Java (Android)

这些 API 用于与 Android 的 Java 运行时进行交互。所有 Java 相关代码都必须包裹在 `Java.perform(function() { ... });` 中。

API/代码片段	描述
<code>Java.perform(function() { ... });</code>	Frida 中与 Java 交互的入口点和作用域
<code>Java.available</code>	检查 Java VM 是否可用
<code>var MyClass = Java.use('com.example.MyClass');</code>	获取一个类的包装器，用于方法 Hook 或创建实例
<code>MyClass.myMethod.implementation = function(...) { ... }</code>	替换 (Hook) 一个方法的实现
<code>this.myMethod(...)</code>	在 Hook 的实现中调用原始方法
<code>MyClass.\$new()</code>	创建一个类的新实例
<code>Java.choose('com.example.MyClass', { onMatch: ..., onComplete: ... })</code>	查找堆上特定类的所有活动实例
<code>Java.cast(obj, MyClass)</code>	将一个对象转换为特定的类类型
<code>Java.backtrace(this.context, true)</code>	获取当前线程的 Java 调用堆栈
<code>send(data)</code>	从脚本向 Python/Node.js 工具发送消息
<code>recv(callback)</code>	从 Python/Node.js 工具接收消息

Objective-C (iOS)

这些 API 用于与 iOS 的 Objective-C 运行时进行交互。

API/代码片段	描述
<code>ObjC.classes.MyClass</code>	获取一个类的引用
<code>Interceptor.attach(ObjC.classes.MyClass['- myMethod'], { ... })</code>	附加到方法的实现 (Native Interceptor)
<code>ObjC.choose(ObjC.classes.MyClass, { ... })</code>	查找特定类的所有活动实例
<code>ObjC.available</code>	检查 Objective-C 运行时是否可用

通用/原生 (Native)

这些 API 用于与原生代码 (C/C++) 进行交互，跨平台通用。

API/代码片段	描述
<code>Interceptor.attach(ptr("..."), { onEnter: ..., onLeave: ... })</code>	拦截指定地址的原生函数调用
<code>Module.findExportByName("libname.so", "function_name")</code>	按名称查找模块（库）的导出函数地址
<code>Module.findBaseAddress("libname.so")</code>	获取模块加载的基地址
<code>Memory.readByteArray(address, size)</code>	从指定地址读取字节数组
<code>Memory.writeByteArray(address, bytes)</code>	向指定地址写入字节数组
<code>NativeFunction(address, returnType, argTypes)</code>	创建一个可调用的原生函数对象
<code>ptr("0x...")</code>	创建一个原生指针
<code>Thread.backtrace(this.context, Backtracer.ACCURATE)</code>	获取当前线程的原生调用堆栈

JNI (Java Native Interface)

JNI 是 Android 逆向中的重要组成部分，Frida 提供了强大的 JNI Hook 能力。

API/代码片段	描述
<code>Module.findExportByName("lib.so", "Java_com_pkg_Class_method")</code>	查找 JNI 函数地址
<code>Java.vm.getEnv()</code>	获取当前线程的 JNIEnv 指针
<code>Java.vm.tryGetEnv()</code>	尝试获取 JNIEnv (不会抛异常)
<code>Java.vm.perform(callback)</code>	在 Java 虚拟机线程中执行回调

常用脚本示例

JNI Hook 示例

```
// Hook JNI Function
var jni_func = Module.findExportByName(
    "libnative.so",
    "Java_com_example_app_Crypto_encrypt"
);

if (jni_func) {
    Interceptor.attach(jni_func, {
        onEnter: function (args) {
            console.log("[JNI Hook] encrypt() called");

            // args[0] = JNIEnv*
            // args[1] = jclass/jobject
            // args[2] = first Java parameter

            // Read jstring parameter
            if (args[2]) {
                var env = Java.vm.getEnv();
                var jstr = args[2];
                var cstr = env.getStringUtfChars(jstr, null);
                console.log("Input: " + cstr.readCString());
                env.releaseStringUtfChars(jstr, cstr);
            }
        },
        onLeave: function (retval) {
            // Read returned jstring
            if (retval && !retval.isNull()) {
                var env = Java.vm.getEnv();
                var cstr = env.getStringUtfChars(retval, null);
                console.log("Output: " + cstr.readCString());
                env.releaseStringUtfChars(retval, cstr);
            }
        },
    });
}

// Also hook the native method call from Java layer
Java.perform(function () {
    var Crypto = Java.use("com.example.app.Crypto");

    Crypto.encrypt.implementation = function (input) {
        console.log("[Java Hook] encrypt called with: " + input);
        var result = this.encrypt(input);
        console.log("[Java Hook] encrypt returned: " + result);
        return result;
    };
});
```

枚举 JNI 函数

```
function enumerateJNIFunctions(moduleName) {
    var module = Process.getModuleByName(moduleName);
    var exports = module.enumerateExports();

    console.log("[JNI Enumeration] " + moduleName);
    exports.forEach(function (exp) {
        if (exp.name.startsWith("Java_")) {
            console.log(" " + exp.name + " @ " + exp.address);
        }
    });
}

// Usage example
enumerateJNIFunctions("libnative.so");
```

Hook Java 方法

```
Java.perform(function () {
    var MyClass = Java.use("com.example.SecretClass");

    MyClass.secretMethod.implementation = function (arg1, arg2) {
        console.log("secretMethod called with:", arg1, arg2);

        // Call original method and get return value
        var retval = this.secretMethod(arg1, arg2);
        console.log("Original return value:", retval);

        return retval; // Return original value
    };
});
```

绕过 VIP 检查

```
Java.perform(function () {
    var PremiumUtils = Java.use("com.example.PremiumUtils");

    PremiumUtils.isUserPremium.implementation = function () {
        console.log("Bypassing isUserPremium check...");
        return true; // Always return true to bypass VIP check
    };
});
```

查找堆上的实例

```
Java.perform(function () {
    Java.choose("com.example.UserManager", {
        onMatch: function (instance) {
            console.log("Found UserManager instance:", instance);
            console.log("User ID:", instance.getUserId());
        },
        onComplete: function () {
            console.log("Search complete.");
        },
    });
});
```

Hook 构造函数

```
Java.perform(function () {
    var User = Java.use("com.example.User");

    User.$init.implementation = function (name, age) {
        console.log("User object created with name:", name, "and age:", age);

        // Call original constructor
        this.$init(name, age);
    };
});
```

Hook Native 函数 (SSL_write)

```
var ssl_write = Module.findExportByName("libssl.so", "SSL_write");

Interceptor.attach(ssl_write, {
    onEnter: function (args) {
        // args[0] is the SSL context
        // args[1] is the buffer
        // args[2] is the size
        console.log("Intercepted SSL_write, size:", args[2].toInt32());
        // You can use hexdump(args[1]) to view the data
    },
    onLeave: function (retval) {
        // retval is the original return value
        console.log("SSL_write returned:", retval.toInt32());
    },
});
```

RPC 导出函数

JavaScript 脚本 (script.js):

```
function getSecretValueFromApp() {
    var secret = "";
    Java.perform(function () {
        // Assume there's a method to get the secret value
        var Utils = Java.use("com.example.Utils");
        secret = Utils.getSecret();
    });
    return secret;
}

// Export function
rpc.exports.getsecret = getSecretValueFromApp;
```

Python 调用:

```
import frida

# ... Connect to device and attach to process ...
# script = session.create_script(js_code)
# ...
# script.load()

# Call the exported function from the script
secret = script.exports.getsecret()
print("Secret from app:", secret)
```

[T02] Frida 内部原理

T02: Frida 核心模块与实现原理

Frida 是一个功能强大的动态插桩框架，但要充分利用它，理解其内部工作原理至关重要。本指南将深入探讨构成 Frida 的几个核心模块、它们的作用以及它们是如何协同工作的。

目录

- [Frida 核心模块与实现原理](#)
 - [目录](#)
 - [Frida 的 architecture 概览](#)
 - [核心组件详解](#)
 - [Frida-Server: 设备端的守护进程](#)
 - [Frida-Core: 注入目标进程的核心引擎](#)
 - [Frida-Gum: 实现 Hook 的魔法棒](#)
 - [Interceptor : 函数拦截器](#)
 - [Stalker : 指令级跟踪器](#)
 - [JavaScript \(V8\) 运行时: 脚本的执行环境](#)
 - [语言绑定 \(Bindings\): 你的控制台](#)
 - [工作流程串讲](#)

Frida 的 architecture 概览

Frida 采用的是一种客户端-服务器 (Client-Server) 架构。

!!! question "思考：为什么需要这样复杂的架构？" Frida 为什么不设计成一个简单的工具，而要分成客户端、服务器、Agent 三层？

- 跨平台的必然选择：
- 隔离性：你的分析脚本（Python）运行在 PC，不会影响目标设备的性能
- 安全性：Server 只负责进程管理和注入，真正的“危险操作”在隔离的进程中内
- 灵活性：同一个 Server 可以同时为多个客户端服务，支持团队协作
- 跨语言：PC 端用 Python/Node.js 编写自动化脚本，目标进程内用 JavaScript 操作内存，各取所长

这种架构的本质是：把“控制”和“执行”分离，就像遥控无人机——遥控器在你手上，但飞行逻辑在机上。

- 客户端（Client）：运行在你 PC 上的部分。这包括你编写的 Python 或 Node.js 脚本，以及你使用的 Frida 命令行工具（`frida`, `frida-trace` 等）。
- 服务器（Server）：在目标设备（如 Android 手机）上以后台守护进程模式运行的 `frida-server`。
- Agent：当你附加到一个目标进程时，Frida 会将一个动态库（`frida-agent.so`）注入到该进程的内存空间中。这个 Agent 负责执行你在客户端脚本中定义的逻辑。

Frida Architecture

- 图片来源: [frida.re*](#)
-

核心组件详解

Frida-Server: 设备端的守护进程

`frida-server` 是一个在目标设备上运行的二进制文件。它的主要职责是：

1. 监听连接：监听来自你 PC 上 Frida 客户端的 TCP 连接。
 2. 进程管理：枚举目标设备上正在运行的进程，获取应用信息。
-

3. 注入 Agent: 当客户端指定要附加 (attach) 或启动 (spawn) 一个应用时, `frida-server` 负责将 `frida-agent.so` 注入到目标进程中。在 Android 上, 它通常通过 `ptrace` 来实现这一点。

Frida-Core: 注入目标进程的核心引擎

`frida-core` 是 Frida 的核心, 它被编译成 `frida-agent.so` 并注入到目标进程。它是一个用 C 语言编写的多平台库, 主要负责:

1. 进程内通信: 建立一个与 `frida-server` 的通信渠道, 从而间接地与你的 PC 客户端通信。
2. 加载 JavaScript 引擎: 它内部嵌入了一个 Google V8 JavaScript 引擎。
3. 暴露原生 API: 将底层的 `frida-gum` 功能通过 JavaScript API (如 `Interceptor`, `Memory`, `NativePointer`) 暴露给用户脚本。

Frida-Gum: 实现 Hook 的魔法棒

`frida-gum` 是 `frida-core` 中最具魔力的部分, 它是一个跨平台的代码插桩工具包。所有 Hook 和代码跟踪功能都由它提供。

Interceptor : 函数拦截器

`Interceptor` 是你最常使用的功能, 用于 Hook/Trace/替换任意函数。

!!! tip "深入理解: Hook 的本质是什么?" 很多人把 Hook 当成"黑魔法", 但其实原理很朴素:

- Hook = 劫持程序的执行流

想象你在高速公路上设置了一个收费站:

1. 原始道路: 函数的正常执行流程
2. 收费站 (Trampoline) : 你插入的代码
3. 改道标志 (JMP) : 修改函数入口的跳转指令
4. 恢复通行: 执行原始指令后继续

理解了这个本质, 你就能:

- 判断哪些 Hook 会相互冲突 (都修改同一个函数入口)

- 理解为什么有些反 Hook 检测能发现你（检查函数头的修改）
- 知道如何写更隐蔽的 Hook (inline hook vs. PLT/GOT hook)
- 实现原理:

1. 动态代码生成: 当你 `Interceptor.attach` 一个函数时, Frida-Gum 会在内存中动态地生成一小段汇编代码, 我们称之为蹦床 (Trampoline)。
2. 函数头重写 (Prologue Rewriting): Frida-Gum 会修改目标函数入口点 (函数头) 的几条指令, 将其替换为一个无条件跳转 (`JMP`) 指令, 该指令指向刚刚创建的蹦床。Frida 会非常小心地保存被它覆盖掉的原始指令。
3. 执行流程:

- 当应用调用目标函数时, 它会首先跳转到蹦床。
- 蹦床代码会保存当前的 CPU 上下文 (寄存器状态), 然后调用你在 JavaScript 中定义的 `onEnter` 回调。
- `onEnter` 执行完毕后, 蹦床会执行被它覆盖掉的原始函数指令, 然后跳转回原始函数的剩余部分继续执行。
- 当原始函数执行完毕后, 控制权返回给蹦床, 蹦床再调用你的 `onLeave` 回调。
- 最后, 蹦床恢复之前保存的 CPU 上下文, 并将返回值传递给原始的调用者。

Stalker : 指令级跟踪器

`Stalker` 是 Frida 的代码跟踪引擎, 功能极其强大但使用也更复杂。它可以用来记录一个线程执行过的每一条汇编指令。

- 实现原理 (基于动态重新编译):
1. 基本块 (Basic Block): Stalker 将代码分解为 “基本块”。一个基本块是一系列连续的指令, 只有一个入口点和一个出口点 (通常是跳转或返回指令)。
 2. 代码拷贝与插桩: 当一个线程将要执行某个基本块时, Stalker 会:
 - a. 将这个基本块的所有指令拷贝到一块新的内存区域。
 - b. 在这份拷贝中插入你的分析代码 (例如, 记录指令地址、寄存器值的代码)。
 - c. 执行这份被插桩后的代码副本。

3. 代码缓存 (Code Cache): Stalker 会缓存这些被修改过的基本块。下次再执行到同一个基本块时，可以直接使用缓存中的版本，极大地提高了性能。

4. 链接 (Chaining): Stalker 会修改每个插桩后基本块的末尾，使其跳转到下一个即将执行的原始基本块对应的“插桩版本”，从而形成一个完整的跟踪链。

简而言之，Stalker 通过创建和执行原始代码的“带监控的副本”来实现无死角的指令级跟踪。

JavaScript (V8) 运行时: 脚本的执行环境

为什么我们用 JavaScript 写 Hook 逻辑？因为 frida-agent.so 在注入目标进程后，会初始化一个 V8 引擎实例。你的 JS 脚本被完整地加载到这个 V8 引擎中执行。

这带来了巨大的优势：

- 高级语言的便利性：你可以在目标进程的地址空间内，用 JavaScript 的便利性来操作内存、调用函数。
- JIT 编译：V8 的即时编译 (JIT) 特性使得你的 JS 脚本能以接近原生的速度运行，性能远超解释执行。
- 强大的生态：可以利用现有的 JS 库。

语言绑定 (Bindings): 你的控制台

frida-python, frida-node 等库是你的“控制端”。它们负责：

- 连接 Server：与设备上的 frida-server 建立通信。
- 发送指令：将你的指令（如“附加到 PID 1234”）发送给 frida-server。
- 加载脚本：将你的 .js 脚本文件内容发送给 frida-agent.so 里的 V8 引擎去执行。
- 双向通信 (RPC)：建立一个双向的 RPC 通道。这使得你在 JS 中调用 send() 的数据能被 Python 的 on_message 回调接收，反之亦然。

工作流程串讲

当你执行 `frida -U -f com.example.app -l script.js` 时，发生了什么？

1. [PC] `frida` (Python 客户端) 解析命令。
2. [PC → Phone] 客户端通过 USB 连接到手机上的 `frida-server`。
3. [PC → Phone] 客户端向 `frida-server` 发送指令：“请以 `spawn` 模式启动 `com.example.app`”。
4. [Phone] `frida-server` 找到 `com.example.app` 并启动它，但使其处于暂停状态。
5. [Phone] `frida-server` 将 `frida-agent.so` 注入到这个新创建的应用进程中。
6. [Phone] `frida-agent.so` 在进程内初始化，启动 V8 引擎，并建立与 `frida-server` 的内部通信。
7. [PC → Phone] 客户端读取 `script.js` 的内容，并通过 `frida-server` 将其发送给 `frida-agent.so`。
8. [Phone] `frida-agent.so` 中的 V8 引擎执行 `script.js` 的代码（例如，`Interceptor.attach(...)`）。
9. [PC → Phone] 客户端发送“恢复进程”的指令。
10. [Phone] 应用进程从暂停状态中恢复，开始正常执行。当它调用被 Hook 的函数时，你在 `script.js` 中定义的逻辑就会被触发。
11. [双向] 脚本中的 `send()` 消息会通过 `agent → server → client` 的路径回到你的 PC 终端上显示。

[T03] Xposed 使用指南

T03: Xposed 框架入门指南

Xposed 是一个在 Android 平台上广受欢迎的动态代码 Hook 框架。与 Frida 主要用于实时、临时的分析不同，Xposed 旨在对系统和应用进行永久性的修改。它通过替换一个核心系统进程 (`app_process`)，在应用启动时加载自定义模块，从而实现对任意方法的高效 Hook。

目录

- [Xposed 框架入门指南](#)
 - [Xposed 是一个在 Android 平台上广受欢迎的动态代码 Hook 框架。与 Frida 主要用于实时、临时的分析不同，Xposed 旨在对系统和应用进行永久性的修改。它通过替换一个核心系统进程 \(`app_process`\)，在应用启动时加载自定义模块，从而实现对任意方法的高效 Hook。](#)
 - [目录](#)
 - [核心原理](#)
 - [Xposed vs. Frida](#)
 - [环境搭建 \(以 LSPosed 为例\)](#)
 - [开发第一个 Xposed 模块](#)
 - [1. 项目结构](#)

核心原理

Xposed 的工作基础是它能够在 Android 系统启动的核心阶段介入，并将自己的代码注入到每一个应用程序进程中。

1. Zygote 注入: Xposed 通过替换系统原生的 `/system/bin/app_process` 可执行文件，实现了对 Zygote 进程（所有 App 进程的父进程）的控制。当 Zygote 启动时，会加载 Xposed 的核心 Jar 包 (Xposed Bridge) 。

2. 方法 Hook: 当模块需要 Hook 一个方法时, Xposed 会在运行时深入虚拟机 (ART) 内部, 直接修改该方法在内存中的数据结构。它将目标方法"伪装"成一个 Native 方法, 并将其执行入口指向 Xposed 的一个通用桥接函数。
3. 执行流重定向: 当 App 调用被 Hook 的方法时, 执行流会先进入 Xposed 的桥接函数, 在这里 Xposed 依次调用所有模块的 `beforeHookedMethod`, 然后调用原方法, 最后再调用所有模块的 `afterHookedMethod`, 从而实现对方法调用的完全控制。

想要更深入地了解其实现细节, 请参考 [Xposed Internals: A Deep Dive](#)。

Xposed vs. Frida

特性	Xposed	Frida
核心目标	永久性修改: 对应用或系统功能进行长期、稳定的修改。	动态分析: 用于实时、临时的分析、逆向和快速原型验证。
运行环境	需要 Root, 通过刷入框架修改系统, 需要重启。	通常需要 Root, 但无需重启, 通过 <code>frida-server</code> 动态附加。
开发语言	Java: 模块是标准的 Android APK。	JavaScript: 主要使用 JS 编写脚本, 也支持其他语言绑定。
开发周期	较慢: 编码 → 编译 APK → 安装 → 激活 → 重启 App/设备 → 测试。	极快: 编写/修改脚本 → 附加进程 → 立即看到结果。
稳定性	极高。为长期运行设计, 一旦激活, 随 App 启动自动生效。	较低。依赖于 <code>frida-server</code> 和附加会话, App 重启后失效。
适用场景	UI 定制、功能增强、去广告、隐私控制(如伪造数据)。	SSL Pinning 绕过、算法逆向、协议分析、漏洞挖掘。

- 总结: 如果你想写一个"插件"来永久性地改变一个 App 的功能, 用 Xposed; 如果你想分析一个 App 的内部行为, 用 Frida。

环境搭建 (以 LSPosed 为例)

当前, LSPosed 是社区中最主流、兼容性最好的 Xposed 框架实现, 它基于 Riru/Zygisk, 以 Magisk 模块的形式工作。

1. 前提条件:

- 一台已解锁并刷入 Magisk 的 Android 设备 (Android 8.1+) 。

1. 安装 Riru 或启用 Zygisk:

- Zygisk (推荐): 在 Magisk Manager 中, 进入设置, 开启 `Zygisk` 选项。

- Riru (备选): 在 Magisk Manager 的"模块"部分, 搜索并安装 `Riru` 模块。

1. 安装 LSPosed:

- 从 [LSPosed 的 GitHub Releases](#) 页面下载最新的 Zygisk 版本 ZIP 包。

- 在 Magisk Manager 的"模块"页, 选择"从本地安装", 然后选中下载的 LSPosed ZIP 包。

- 安装完成后, 点击右下角的"重启"按钮。

1. 验证安装:

- 重启后, 桌面上会出现 LSPosed 的管理程序图标。

- 打开 LSPosed, 如果状态显示为"已激活", 则表示框架安装成功。

开发第一个 Xposed 模块

我们将创建一个简单的模块, 来 Hook 系统的时钟, 在后面加上一个小尾巴。

1. 项目结构

- 在 Android Studio 中创建一个新的、空的 Android 项目。

- 在 app 的 `build.gradle` 文件中添加 Xposed API 依赖:

```
dependencies {  
    // ... other dependencies  
    compileOnly 'de.robv.android.xposed:api:82'  
    // 'compileOnly' is used because the framework is already provided by the system,  
    only needed at compile time  
}
```

- 创建一个新的 Java 类，例如 `ClockHook`，并让它实现 `IXposedHookLoadPackage` 接口。

```
package com.example.myxposedmodule;

import de.robv.android.xposed.IXposedHookLoadPackage;
import de.robv.android.xposed.XC_MethodHook;
import de.robv.android.xposed.XposedBridge;
import de.robv.android.xposed.XposedHelpers;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;
import android.widget.TextView;

public class ClockHook implements IXposedHookLoadPackage {
    @Override
    public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {
        // We only care about system UI
        if (!lpparam.packageName.equals("com.android.systemui")) {
            return;
        }

        XposedBridge.log("Loaded app: " + lpparam.packageName);

        // Find and Hook the clock update class and method
        // (Note: class names and method names may differ across Android versions)
        XposedHelpers.findAndHookMethod(
            "com.android.systemui.statusbar.policy.Clock", // Class name
            lpparam.classLoader, // ClassLoader
            "updateClock", // Method name
            new XC_MethodHook() { // Hook callback
                @Override
                protected void afterHookedMethod(MethodHookParam param) throws Throwable {
                    super.afterHookedMethod(param);
                    TextView clockView = (TextView) param.thisObject;
                    String originalTime = clockView.getText().toString();
                    String newTime = originalTime + " ";
                    clockView.setText(newTime);
                    XposedBridge.log("Clock hooked! New time: " + newTime);
                }
            }
        );
    }
}
```

- 在 `app/src/main/AndroidManifest.xml` 的 `<application>` 标签内，添加 meta-data 来声明这是一个 Xposed 模块。

```
<meta-data  
    android:name="xposedmodule"  
    android:value="true" />  
<meta-data  
    android:name="xposeddescription"  
    android:value="This is an example module that adds a tail to system clock" />  
<meta-data  
    android:name="xposedminversion"  
    android:value="52" />
```

- 在 `assets` 文件夹内，创建一个名为 `xposed_init` 的文本文件。
- 在 `xposed_init` 文件中，写入你的 Hook 类的完整路径：

```
com.example.myxposedmodule.ClockHook
```

1. 构建 APK: 在 Android Studio 中构建你的项目，生成 APK。

2. 安装 APK: 将 APK 安装到你的测试设备上。

3. 激活模块：

- 打开 LSPosed Manager。
- 进入"模块"部分，找到你刚刚安装的模块。
- 点击它，然后启用模块。
- 在作用域列表中，勾选"SystemUI"。

1. 重启目标进程：

- 在 LSPosed 的状态页右上角，点击三个点菜单，选择"软重启"或"重启 SystemUI"，或者直接重启手机。

1. 查看效果：查看你的状态栏时钟，它现在应该带有一个小尾巴了！你也可以在 LSPosed 的日志中看到 `XposedBridge.log` 输出的信息。

核心 API 详解

IXposedHookLoadPackage

这是所有模块的入口点。它只有一个方法 `handleLoadPackage(LoadPackageParam lpparam)`。当任何一个 App 启动时, Xposed 都会调用这个方法, 并传入 `lpparam` 对象, 其中包含了非常有用的信息:

- `lpparam.packageName` : 当前加载的 App 的包名。
- `lpparam.processName` : 当前进程名。
- `lpparam.classLoader` : 当前 App 的 ClassLoader, 这是 Hook App 内部类的必需品。

XposedHelpers

一个包含大量静态辅助方法的工具类, 极大简化了反射操作。

- `findAndHookMethod(String className, ClassLoader classLoader, String methodName, Object... parameterTypesAndCallback)` : 最核心的 Hook 方法。最后一个参数必须是 `XC_MethodHook` 回调。
- `findClass(String className, ClassLoader classLoader)` : 查找一个类。
- `getObjectContext(Object obj, String fieldName) / setObjectContext(Object obj, String fieldName, Object value)` : 获取/设置对象的成员变量。
- `callMethod(Object obj, String methodName, Object... args)` : 调用一个对象的方法。
- `getStaticObjectField(...)` / `callStaticMethod(...)` : 用于操作静态变量和静态方法。

XC_MethodHook

这是一个抽象类, 你需要继承它并重写它的两个核心方法。

- `beforeHookedMethod(MethodHookParam param)` : 在原方法执行前被调用。
- `afterHookedMethod(MethodHookParam param)` : 在原方法执行后被调用。

这两个方法都接收一个 `MethodHookParam` 对象，它包含了本次方法调用的所有上下文信息：

- `param.thisObject` : `this` 指针，即方法所属的对象实例。
 - `param.args` : `Object[]` 数组，包含了方法被调用时的所有参数。你可以在 `beforeHookedMethod` 中修改它。
 - `param.getResult()` : 获取原方法的返回值。只能在 `afterHookedMethod` 中调用。
 - `param.setResult(Object result)` : 设置一个新的返回值。如果在 `beforeHookedMethod` 中调用，原方法将不会被执行。如果在 `afterHookedMethod` 中调用，它会覆盖原方法的返回值。
 - `param.getThrowable()` / `param.setThrowable(Throwable t)` : 用于获取/设置方法抛出的异常。
-

常见应用场景

- UI 定制: 修改系统或应用的外观，如状态栏、通知、锁屏等（代表作：`GravityBox`）。
 - 功能增强: 为应用添加原生不支持的功能，如为微信添加防撤回、自动抢红包功能。
 - 去除限制: 破解应用的付费功能、去除截图限制、去除广告等。
 - 隐私保护: 拦截应用获取敏感信息的请求（如定位、联系人、设备ID），并返回虚假或空数据（代表作：`XPrivacyLua`）。
 - 安全分析:
 - 绕过 SSL Pinning（尽管 Frida 更灵活）。
 - 禁用 Root 检测或反调试机制。
 - 日志记录：打印关键方法的参数和返回值，分析应用行为。
-

[T04] Xposed 内部原理

T04: Xposed 内部原理深度剖析

Xposed 是一个强大的 Android 框架，允许用户在运行时修改系统和应用程序进程的行为，而无需修改任何 APK 文件。本文档深入探讨了 Xposed 工作的核心原理。

1. 入口点：Zygote 进程注入

Xposed 的基础在于它能够将自定义代码注入到每个 Android 应用程序进程中。它通过针对 Zygote 进程来实现这一点，Zygote 是 Android OS 中的原始进程，所有应用程序进程都从它 fork 而来。

工作原理：

1. 替换 `app_process`：在安装期间，Xposed 用自己修改后的版本替换原始的 `/system/bin/app_process` 可执行文件。这个可执行文件是 Zygote 进程启动的第一个程序。
2. 加载桥接器：当 Zygote 启动时，它运行 Xposed 版本的 `app_process`。这个自定义可执行文件的主要任务是将一个特殊的 JAR 文件（通常称为 Xposed Bridge，即 `XposedBridge.jar`）加载到 Zygote 的地址空间中。
3. 通过 Fork 继承：由于每个 Android 应用程序都是 Zygote 进程的 fork，它们都继承了父进程的内存空间。这意味着 Xposed Bridge JAR 从创建时刻起就自动加载到每个应用程序进程中。

这种巧妙的方法确保 Xposed 的核心逻辑在任何应用程序中都存在并准备好执行，为方法 Hook 提供了一个通用平台。

2. 核心魔法：方法 Hook

Xposed 最著名的功能是其"Hook"Java 方法的能力。这不是简单的反射；它是对底层虚拟机数据结构的深度操纵。

Method 结构转换：

核心思想是改变虚拟机中目标 Java 方法的类型签名，使虚拟机认为它是一个 `native` 方法。

1. 查找目标：模块使用 `findAndHookMethod` 等辅助函数来指定它们希望 Hook 的类和方法。
2. 修改 `Method` 对象：在内部，Xposed 使用反射和本地代码来获取与目标对应的 Java `java.lang.reflect.Method` 对象的句柄。
3. "Native"伪装：

- Xposed 修改 `Method` 对象的 `accessFlags`，添加 `ACC_NATIVE` 标志。
- 然后它覆盖方法的入口点指针。在 ART 运行时中，这意味着替换内部 `ArtMethod` 结构中的 `entry_point_from_quick_compiled_code_` 字段。
- 这个新的入口点现在指向 Xposed 提供的通用原生桥接函数。

1. 保存原始方法：在覆盖之前，Xposed 仔细地将原始方法的信息（包括其原始入口点和访问标志）保存到单独的备份结构中。

Hooked 方法的执行流程：

当应用程序调用被 Xposed Hook 的方法时，会发生以下序列：

1. 绕道到原生桥接器：虚拟机现在认为该方法是原生方法，将调用定向到 Xposed 的通用原生桥接函数。
2. 回调到 Java 桥接器：原生函数做的事情很少。它的主要目的是回调到 Java 世界，调用 Xposed Bridge 中的核心 Java 方法：`handleHookedMethod`。
3. `handleHookedMethod` 协调：这个强大的 Java 方法协调整个 Hook 生命周期：
 - a. 它将方法的参数和 `this` 引用准备到一个 `MethodHookParam` 对象中。
 - b. `beforeHookedMethod`：它遍历模块中所有注册的回调，并调用它们的 `beforeHookedMethod` 方法。这些回调可以检查或修改参数。关键的是，“before”回调可以选择通过直接在 `param` 对象上设置结果来完全跳过原始方法。

c. 调用原始方法：如果方法没有被跳过，`handleHookedMethod` 使用保存的备份信息来调用原始方法，并传入（可能已修改的）参数。

d. `afterHookedMethod`：在原始方法完成后，它再次遍历回调，这次调用它们的`afterHookedMethod`方法。这些回调可以检查或修改方法的返回值。

1. 返回给调用者：最后，`handleHookedMethod` 将最终结果（来自“before”回调或原始方法的（已修改的）结果）返回给应用程序的原始调用点。

整个过程对应用程序代码是透明的，应用程序只是看到一个返回值的方法调用，而不知道它经历了复杂的绕道。

3. 模块加载机制

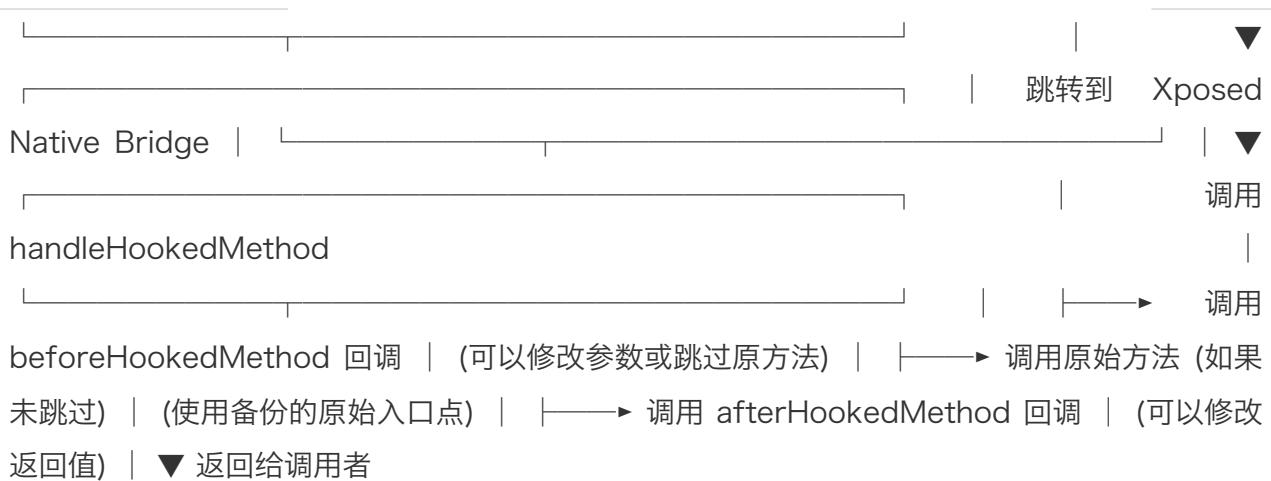
Xposed 模块是标准的 Android APK，它们向框架表明自己的性质。

- `AndroidManifest.xml`：模块的清单文件必须包含一个`<meta-data>` 标签，其中`android:name="xposedmodule"` 设置为`true`。
- `assets/xposed_init`：模块`assets` 目录中的这个文件是一个简单的文本文件。每行指向一个完全限定的类名。
- `IXposedHookLoadPackage`：`xposed_init` 中列出的类必须实现这个接口。Xposed 框架将实例化这些类，并为每个加载的应用程序包调用它们的`handleLoadPackage` 方法，允许模块决定是否应用其 Hook。

架构图解

Xposed 工作流程图





**关键字段修改: **

- `access_flags_`: 添加 `ACC_NATIVE` 标志
- `entry_point_from_quick_compiled_code_`: 替换为 Xposed 桥接函数地址
- 备份原始字段值以便后续恢复

**伪代码示例: **

```
```cpp
// Xposed 内部简化逻辑
void hookMethod(ArtMethod* method) {
 // 保存原始信息
 backup.original_flags = method->access_flags_;
 backup.original_entry = method->entry_point_from_quick_compiled_code_;

 // 修改为 native 方法
 method->access_flags_ |= ACC_NATIVE;
 method->entry_point_from_quick_compiled_code_ = xposed_bridge_entry;
}
```

模块 A - beforeHookedMethod | ▼ 模块 B - beforeHookedMethod | ▼ 原始方法执行  
| ▼ 模块 B - afterHookedMethod | ▼ 模块 A - afterHookedMethod | ▼ 返回结果

### ### 3. 性能优化机制

#### \* \*JIT/AOT 编译影响: \*\*

- Hooked 方法被标记为 native, 避免 JIT 编译
- 通过 native 桥接的额外开销 (约 10–50μs 每次调用)
- 大量 Hook 可能影响应用启动时间

#### \* \*最佳实践: \*\*

- 只 Hook 必要的方法
- 在回调中避免耗时操作
- 使用条件判断减少不必要的处理

#### ## 与其他 Hook 框架对比

特性	Xposed	Frida	VirtualXposed
**需要 Root**	是	否 (Gadget 模式除外)	否
**注入方式**	Zygote 级别	进程级别	虚拟化容器
**性能开销**	低–中	中–高	中
**开发语言**	Java	JavaScript/Python	Java
**动态性**	重启应用生效	实时生效	重启应用生效
**稳定性**	高	中	中–低
**适用场景**	长期修改	动态分析/调试	无 Root 测试

#### ## 安全影响与检测

#### #### 应用层检测方法

##### \* \*1. 检查 Xposed 特征文件: \*\*

```
```java
private boolean isXposedInstalled() {
    try {
        // 检查 XposedBridge 类
        Class.forName("de.robv.android.xposed.XposedBridge");
        return true;
    } catch (ClassNotFoundException e) {
        return false;
    }
}
```

```
int modifiers = method.getModifiers(); return Modifier.isNative(modifiers) && !
shouldBeNative(method); }
```

```
for (StackTraceElement trace : traces) {  
    if (trace.getClassName().contains("XposedBridge")) {  
        return true;  
    }  
}  
return false;  
}
```

1. 清理堆栈跟踪信息
2. 使用定制版 Xposed (修改特征字符串)

实际应用场景

1. 隐私保护

- 伪造设备信息 (IMEI、MAC 地址等)
- 阻止权限请求
- 拦截敏感数据上传

2. 功能增强

- 移除广告
- 解锁 VIP 功能
- 修改应用行为

3. 逆向分析

- 监控方法调用
- 提取加密密钥
- 分析算法逻辑

4. 自动化测试

- 模拟用户行为
 - 注入测试数据
 - 绕过验证码
-

模块开发示例

基础 Hook 示例

```
public class MyXposedModule implements IXposedHookLoadPackage {

    @Override
    public void handleLoadPackage(XC_LoadPackage.LoadPackageParam lpparam)
        throws Throwable {

        // 只 Hook 目标应用
        if (!lpparam.packageName.equals("com.example.target"))
            return;

        // Hook 方法
        findAndHookMethod(
            "com.example.target.MainActivity",
            lpparam.classLoader,
            "getUserInfo", // 方法名
            new XC_MethodHook() {
                @Override
                protected void beforeHookedMethod(MethodHookParam param)
                    throws Throwable {
                    // 在方法执行前
                    XposedBridge.log("getUserInfo 即将被调用");
                }

                @Override
                protected void afterHookedMethod(MethodHookParam param)
                    throws Throwable {
                    // 在方法执行后
                    String result = (String) param.getResult();
                    XposedBridge.log("getUserInfo 返回: " + result);

                    // 修改返回值
                    param.setResult("Fake User Info");
                }
            }
        );
    }
}
```

```
lpparam.classLoader, byte[].class, // 参数类型 String.class, new XC_MethodHook()
{ @Override protected void afterHookedMethod(MethodHookParam param) throws
Throwable { byte[] key = (byte[]) param.args[0]; String algorithm = (String)
param.args[1];

XposedBridge.log("捕获密钥!"); XposedBridge.log("算法: " + algorithm);
XposedBridge.log("密钥: " + bytesToHex(key)); } } );
```

1. 依赖 Root 权限：需要系统级访问
2. 稳定性问题：不当使用可能导致系统崩溃
3. 版本兼容性：需要针对不同 Android 版本适配
4. 检测与对抗：越来越多应用实施反 Xposed 检测

未来趋势

1. EdXposed/LSPosed：基于 Riru/Zygisk 的新实现
2. 虚拟化方案：VirtualXposed、太极等无需 Root 的方案
3. 对抗升级：更复杂的检测与反检测技术

总结

Xposed 通过以下核心技术实现了强大的运行时修改能力：

1. Zygote 注入：确保每个应用都加载 Xposed Bridge
2. 方法伪装：将 Java 方法转换为 native，重定向入口点
3. 回调机制：在方法执行前后插入自定义逻辑
4. 模块化设计：灵活的 APK 插件系统

这种设计使 Xposed 成为 Android 平台上最强大的运行时修改框架之一，但同时也带来了安全风险和检测对抗的挑战。

理解 Xposed 的内部原理不仅有助于开发更好的模块，也为逆向工程、安全研究和应用保护提供了重要的技术洞察。

[T05] Unidbg 使用指南

T05: Unidbg 模拟执行框架指南

Unidbg 是一个基于 Java 开发的、开源的、功能强大的 Android/iOS 原生库 (`.so` / `.dylib`) 模拟执行框架。它能够在 PC (Windows/Linux/macOS) 上模拟一个完整的 ARM 执行环境，使得你可以像调用本地 Java 方法一样直接调用和调试原生库中的函数。这对于分析高度混淆、包含大量环境依赖和反调试机制的原生算法来说，是一个革命性的工具。

目录

- 核心思想与应用场景
- Unidbg vs. Frida
- 环境搭建
- 基本使用流程
- 实战技巧

核心思想与应用场景

Unidbg 的核心思想是“欺骗”。它通过以下方式让 `.so` 文件认为自己正运行在一个真实的 Android 设备上：

- 模拟文件系统：创建一个虚拟的文件系统，你可以将应用的数据、配置文件等放入其中。
- 模拟内存空间：加载 `.so` 文件及其依赖的系统库（如 `libc.so`, `libdl.so`）到模拟的内存空间中。
- 模拟 JNI 环境：实现了大部分 JNI 函数，当 `.so` 文件试图通过 JNI 调用 Java 层代码时，Unidbg 会拦截并返回你指定的值。

- Hook 系统调用 (SVC): 拦截底层的系统调用，返回预设的结果。

主要应用场景

- 算法复现 (一把梭): 直接调用目标加密/解密函数，输入参数并获取返回值，无需费力去逆向算法本身。
 - 绕过环境检测: 目标函数可能包含对 Root、模拟器、设备 ID 等的检测。Unidbg 可以轻松 Hook 这些检测点，让它们全部失效。
 - 绕过反调试: `ptrace` 等反调试手段在 Unidbg 的模拟环境中天然无效。
 - 批量计算/爆破: 编写脚本，批量调用目标函数，用于参数的爆破或生成大量签名。
 - 主动调用非导出函数: 与 Frida 不同，只要知道函数偏移，就可以直接调用任何函数，无论它是否被导出。
-

Unidbg vs. Frida

特性	Unidbg	Frida
执行环境	PC 端 (模拟执行)	移动设备端 (真机/模拟器)
工作模式	将 <code>.so</code> 当作一个"黑盒"库来调用	侵入正在运行的应用进程进行 Hook
依赖	仅需要 <code>.so</code> 文件及其依赖的库	需要一个完整的、能运行的 APK
反调试	天然免疫	需要编写脚本来对抗反调试
环境依赖	需要手动模拟或 Hook	运行在真实环境中，无需模拟
性能	较低 (因为是全模拟)	较高 (代码在设备上原生运行)
适用性	适合纯算法分析，不涉及 UI 和复杂业务流	适合分析与 Android 系统、UI 强相关的业务逻辑

环境搭建

1. JDK: 确保已安装 JDK 8 或更高版本。
2. Maven: 用于项目构建和依赖管理。
3. IDE: 推荐使用 IntelliJ IDEA。
4. 下载 Unidbg: 从其 GitHub Release 页面下载最新的发行版 `unidbg-dist.zip`，或直接使用 Maven 依赖。
5. 创建 Maven 项目: 在 IDE 中创建一个新的 Maven 项目，并在 `pom.xml` 中添加 Unidbg 的依赖:

```
<dependency>
    <groupId>com.github.unidbg</groupId>
    <artifactId>unidbg-android</artifactId>
    <version>0.9.7</version> <!-- Use latest version -->
</dependency>
```

基本使用流程

以下是一个调用 `.so` 中简单函数的典型代码结构：

```
import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Module;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.*;
import com.github.unidbg.memory.Memory;

import java.io.File;

public class MyTest extends DvmObject<String> {

    private final AndroidEmulator emulator;
    private final Module module;
    private final VM vm;

    public MyTest() {
        // 1. Create emulator instance
        emulator = AndroidEmulatorBuilder.for32Bit().build();
        final Memory memory = emulator.getMemory();
        // 2. Set system library resolver
        memory.setLibraryResolver(new AndroidResolver(23)); // API 23

        // 3. Create virtual machine instance (DVM)
        vm = emulator.createDalvikVM(new File("target/classes/apk/app-debug.apk"));
        vm.setVerbose(true); // Print detailed logs

        // 4. Load target .so
        DalvikModule dm = vm.loadLibrary(new File("target/classes/so/libnative-
lib.so"), true);
        module = dm.getModule();

        // 5. Call JNI_OnLoad (optional but recommended)
        dm.callJNI_OnLoad(emulator);
    }

    public void callNativeFunc() {
        // 6. Call target function
        Number result = module.callFunction(emulator, 0x1234, "hello unidbg") [0];
        System.out.println("Result: " + result.intValue());
    }

    public static void main(String[] args) {
        MyTest test = new MyTest();
        test.callNativeFunc();
        test.emulator.close();
    }
}
```

核心概念

- **VM**: 虚拟机, 可以是 `DalvikVM` (DVM) 或 `ART`。负责管理 Java 对象 (`DvmObject`) 和 JNI 调用。
- **Module**: 代表一个已加载到内存中的 `.so` 模块。
 - `callFunction(emulator, address, args...)`: 核心方法, 通过绝对地址或偏移调用函数。
 - `findSymbolByName("...")`: 按名称查找导出函数。
- **DvmObject**: Java 对象的代理。Unidbg 使用它来向 native 函数传递字符串、字节数组等。
- **AbstractJni**: 如果 `.so` 中有复杂的 JNI 回调, 你需要继承 `AbstractJni` 并重写对应的方法, 以模拟 Java 层的行为。
- Hooking: Unidbg 使用 `com.github.unidbg.hook.Hooker` 接口和 `TraceHook` 等工具来提供类似 Frida 的 Hooking 能力, 可以监控指令、内存读写等。

实战技巧

补环境

如果 `.so` 依赖特定的设备信息或文件, 你需要:

- 在虚拟文件系统中创建对应的文件和内容。
- Hook `open`, `read`, `access` 等 libc 函数, 返回预期的结果。
- Hook JNI 调用, 如 `getSystemService`, 返回一个模拟的 `TelephonyManager` 对象。

定位函数地址

函数地址是 `基地址 (module.base) + 偏移`。偏移可以从 IDA/Ghidra 中获得。

设置断点

使用 `emulator.attach().addBreakPoint(address, ...)` 可以在指定地址设置断点，进行调试。

日志分析

`vm.setVerbose(true)` 会打印非常详细的 JNI 调用和 SVC 日志，这是解决环境问题的关键。

参考官方测试用例

Unidbg 项目的 `unidbg-android/src/test/java` 目录下有大量针对主流 App 的测试用例，是学习 Unidbg 的最佳资料。

[T06] Unidbg 内部原理

T06: Unidbg 实现原理剖析

Unidbg 是一个强大的 Android 原生库（.so）模拟执行框架。它允许你在没有 Android 真机或模拟器的情况下，直接在 PC (macOS, Windows, Linux) 上运行和调试 JNI 函数。理解其工作原理，可以帮助我们更高效地解决复杂的加密算法逆向、协议分析等问题。

目录

1. 核心思想
2. 关键组件详解
3. 工作流程
4. 优势与局限

核心思想

.so 文件是为特定 CPU 架构（如 ARM）和操作系统（如 Android）编译的。它不能直接在你的 x86 架构的 PC 上运行。

Unidbg 的核心思想就是：用纯 Java 在 PC 上构建一个虚拟的、高度仿真的 Android 用户态 (User-Mode) 环境。它不是一个完整的操作系统模拟器，而是专注于模拟一个 Android 进程所需的一切，让 .so 文件“感觉”自己正运行在一个真实的 Android 设备里。

关键组件详解

1. Unicorn Engine - CPU 模拟器

Unicorn 是 Unidbg 的基石。它是一个基于 QEMU 的轻量级、多平台、多架构的 CPU 模拟器库。

方面	说明
作用	负责逐条解释和执行 <code>.so</code> 文件中的 ARM 或 AArch64 (ARM64) 汇编指令
原理	当 Unidbg 加载 <code>.so</code> 的代码段到虚拟内存后，它会设置一个程序计数器 (PC) 指向要执行的函数地址，然后命令 Unicorn 从该地址开始执行。Unicorn 会读取指令、解码、模拟寄存器和内存的读写，并更新 CPU 状态，就像一个真实的 ARM 芯片一样

2. 内存管理与映射 (Memory Management)

Unidbg 内部实现了一套完整的内存管理系统，用于模拟一个进程的虚拟地址空间。

作用：

1. 为加载的 `.so` 文件分配虚拟内存（代码段、数据段、BSS 段等）
2. 管理函数的栈空间 (Stack)，用于存储局部变量和返回地址
3. 处理 `malloc`, `free` 等内存分配请求

原理：

它通过 Java 的数据结构（如 `byte[]` 或 `ByteBuffer`）来表示内存块，并通过一个映射表 (`Map<Long, MemoryBlock>`) 来管理虚拟地址和这些实际内存块之间的关系。当 Unicorn 需要读写某个虚拟地址时，Unidbg 会查询这个表，找到对应的 Java 内存块并进行操作。

3. 动态库加载器 (Dynamic Linker)

Android 应用的 `.so` 文件通常会依赖其他的系统库，如 `libc.so` (标准 C 库), `liblog.so` (日志库), `libz.so` (压缩库) 等。

作用：

Unidbg 内置了一个简易的 `linker`，负责解析 `.so` 文件的依赖项，并加载这些依赖库。

原理：

1. 解析 ELF: Unidbg 会读取 `.so` 文件的 ELF 头，找到其 `.dynamic` section，这里记录了所有依赖库的名称
2. 加载依赖: Unidbg 会在预设的路径中查找这些依赖库（它自带了一些核心的 Android 系统库），然后像加载主 `.so` 一样，将它们也加载到虚拟内存中
3. 符号重定位 (Relocation): 加载器最重要的工作是处理重定位。如果 A.so 调用了 B.so 中的函数 `foo`，A.so 中只存了一个对 `foo` 的“符号引用”。加载器需要在 B.so 中找到 `foo` 的实际地址，然后将这个地址填回到 A.so 的调用指令中。这个过程是 `.so` 文件能够跨库调用的关键

4. 系统调用处理 (Syscall Handler)

当 `.so` 文件需要执行一些需要操作系统内核参与的操作时（如读写文件、网络通信），它会发起一个系统调用 (syscall)，这通过 `SVC` 或 `SWI` 指令实现。

作用：

拦截并处理 `.so` 发出的所有系统调用。

原理：

Unicorn 引擎在执行 `SVC` 指令时会产生一个“中断”，并将控制权交还给 Unidbg。Unidbg 会检查特定的寄存器（如 `r7`）来获取系统调用的编号，然后在其 `SyscallHandler` 中找到对应的 Java 实现并执行。

例如，如果 `.so` 尝试打开一个文件，Unidbg 会拦截这个系统调用，并用 Java 的 `FileInputStream` 在 PC 上实际打开一个文件，然后将文件描述符返回给 `.so`。

5. JNI 函数模拟 (JNI Emulation)

这是 Unidbg 最核心的功能之一。JNI (Java Native Interface) 是 `.so` 文件与 Java 层代码交互的桥梁。

作用：

模拟 Android ART/Dalvik 虚拟机提供的所有 JNI 函数，如 `FindClass`, `GetMethodID`, `CallObjectMethod` 等。

原理：

- Unidbg 在其虚拟环境中预先注册了所有 JNI 函数的 Java 实现
- 当 `.so` 调用 `FindClass("java/lang/String")` 时，Unidbg 的 JNI 模块会接管这个调用，并返回一个代表 `java.lang.String` 类的虚拟对象（一个 Java `DvmClass` 实例）
- 当 `.so` 调用 `callObjectMethod` 时，Unidbg 会根据传入的参数，实际地在 PC 端的 JVM 中执行对应 Java 对象的相应方法，然后将结果返回给 `.so`

通过这种方式，Unidbg 巧妙地将 `.so` 对 Android 虚拟机的调用“嫁接”到了 PC 端的 JVM 上。

工作流程

以下是 Unidbg 运行一个 `.so` 文件的完整流程：

步骤 1：创建模拟器实例

```
AndroidARMEmulator emulator = new AndroidARMEmulator("com.example.app");
```

步骤 2：内存初始化

```
Memory memory = emulator.getMemory();
```

Unidbg 初始化内存管理器和 Unicorn 引擎。

步骤 3：加载动态库

```
Module module = emulator.loadLibrary(new File("libnative-lib.so"));
```

在这一步中：

1. Unidbg 的 `linker` 解析 `libnative-lib.so` 的 ELF 结构
2. 根据 `PT_LOAD` 段，将 `.so` 的内容映射到虚拟内存
3. 解析其依赖库（如 `libc.so`），递归加载它们
4. 进行符号重定位，修复函数调用地址

步骤 4：调用 JNI 函数

```
module.callJNI_OnLoad(emulator);
// 或
DvmObject<?> obj = module.callJniMethod(...);
```

在这一步中：

1. Unidbg 找到目标 JNI 函数在虚拟内存中的地址
2. 设置函数参数，主要是将 `JNIEnv` 和 `jobject` 等 JNI 对象作为指针（虚拟地址）传入
3. 启动 Unicorn 引擎，从目标函数地址开始执行 ARM 汇编指令

步骤 5：执行与交互

在执行过程中：

- 汇编指令由 Unicorn 解释执行
- 遇到系统调用，Unicorn 中断，由 Unidbg 的 `SyscallHandler` 处理
- 遇到调用 JNI 函数，由 Unidbg 的 JNI 模拟层处理，可能会在 PC 的 JVM 上执行真实的 Java 代码

步骤 6：返回结果

函数执行完毕后，从模拟的寄存器（如 `r0`）或栈上获取返回值，并转换为 Java 对象。

优势与局限

优势

优势	说明
摆脱环境限制	无需真机或模拟器，无 root 权限要求
高可控性	可以完全控制程序的执行流程，任意修改内存、寄存器
自动化与集成	易于与 Java/Python 项目集成，进行大规模的自动化测试和分析
反反调试	由于没有实际的调试器进程（ <code>ptrace</code> ），可以绕过大多数基于 <code>ptrace</code> 的反调试检测

局限

局限	说明
环境不完整	并非 100% 完整的 Android 环境。对于强依赖特定系统行为、硬件特性或大量 UI 操作的 <code>.so</code> 文件，模拟可能会失败
性能开销	毕竟是逐条指令模拟，性能远低于原生执行
系统调用和 JNI 覆盖	如果 <code>.so</code> 用到了 Unidbg 尚未实现的系统调用或 JNI 函数，执行会中断，需要手动补充实现

静态分析工具

[T07] Ghidra 入门指南

T07: Ghidra 入门指南

Ghidra 是由美国国家安全局 (NSA) 开发并开源的一款软件逆向工程 (SRE) 套件。它以功能全面、免费开源、跨平台等特性，迅速成为 IDA Pro 之外逆向工程师们的另一个重要选择，尤其在学术界和独立研究者中广受欢迎。

目录

1. 核心特性
 2. Ghidra vs. IDA Pro vs. Radare2
 3. 安装与配置
 4. 基本工作流程
 5. 关键窗口与概念
 - Code Browser (代码浏览器)
 - Decompiler (反编译器)
 - Symbol Tree (符号树)
 - Data Type Manager (数据类型管理器)
 6. 脚本化与自动化
 7. 优缺点分析
-

核心特性

- 强大的反编译器 (Decompiler): 这是 Ghidra 的王牌功能。它内置了一个高质量的、支持多种处理器架构的免费反编译器，能够将汇编代码转换为类似 C/C++ 的高级语言伪代码，极大地提高了代码理解效率。
- 全面的分析能力: 支持对多种平台的可执行文件进行反汇编、分析、反编译、图表绘制和脚本化，包括 Windows, macOS, Linux, Android, iOS 等。
- 强大的脚本引擎: 内置对 Java 和 Python (通过 Jython) 的支持，允许用户编写复杂的脚本来自动化分析任务，从简单的重命名到复杂的漏洞模式匹配。
- 交互式与自动化操作: 既支持像 IDA Pro 那样的交互式手动分析，也提供了强大的"无头分析器"(Headless Analyzer)，可以通过命令行进行批量、自动化的分析。
- 多用户协作: Ghidra Server 组件支持多名分析师对同一个二进制文件进行协同逆向，并能方便地进行版本追踪和合并。
- 可扩展性: 用户可以自定义和扩展 Ghidra 的功能，包括编写新的处理器模块、加载器和分析器插件。

Ghidra vs. IDA Pro vs. Radare2

特性	Ghidra	IDA Pro	Radare2
价格	完全免费	非常昂贵	完全免费
开源	是 (Java)	否	是 (C)
核心优势	高质量的免费反编译器	最强的交互式反汇编	极致的脚本化和命令行
UI	Java Swing, 功能强大但略显笨重	Qt, 业界标准, 成熟稳定	命令行, 或通过 Cutter 提供 GUI
自动化	强大的 Headless 模式和脚本	主要通过 IDC/IDAPython 脚本	设计哲学核心, 自动化能力极强
协作	内置 Ghidra Server 支持	第三方插件 (如 BinSync)	脚本化协作, 或通过第三方工具
学习曲线	中等, UI 直观	中等, 功能繁多	非常陡峭, 命令繁杂

安装与配置

- 前提: 确保已安装 Java Development Kit (JDK) 11 或更高版本。
- 下载: 从 [Ghidra 官方网站](#) 下载最新的稳定版 ZIP 包。
- 解压: 将 ZIP 包解压到任意目录。
- 运行:
 - Windows: 双击运行 `ghidraRun.bat`。
 - Linux / macOS: 在终端中执行 `sh ghidraRun`。

-
5. (可选) Ghidra Dark Theme: Ghidra 的默认主题比较刺眼, 可以通过安装 `Ghidra-dark-theme` 插件来获得更好的视觉体验。
-

基本工作流程

1. 创建项目

- `File -> New Project...`
- 选择 `Non-Shared Project` (单用户项目)。
- 指定项目路径和名称。

2. 导入文件

- `File -> Import File...`
- 选择你想要分析的二进制文件 (如 `.exe`, `.dll`, `.so`, `.apk`)。
- Ghidra 会自动识别文件格式、处理器架构等, 直接点击 `OK`。

3. 分析文件

- 在弹出的分析选项框中, 保留默认勾选的分析器即可, 点击 `Analyze`。
- Ghidra 会开始对文件进行自动分析, 这可能需要一些时间, 取决于文件大小和复杂度。

4. 开始探索

- 分析完成后, 双击项目窗口中的文件名, 打开 Ghidra 的核心工具 `Code Browser`。
 - 现在你可以开始你的逆向之旅了!
-

关键窗口与概念

Code Browser (代码浏览器)

这是 Ghidra 的主界面，通常包含以下几个核心子窗口：

- Listing (清单/反汇编窗口): 左侧显示反汇编代码，是分析的主要区域。
- Functions (函数窗口): 左下角，列出所有已识别的函数。点击函数名可以在反汇编窗口中跳转。
- Program Trees (程序树): 左上角，以树状结构展示程序的段 (sections)。

Decompiler (反编译器)

- 通常位于反汇编窗口的右侧。
- 它会自动显示当前光标所在函数的 C 伪代码。
- 这是 Ghidra 最有价值的窗口。你可以直接在伪代码中对变量、函数进行重命名、修改类型，这些改动会双向同步到反汇编窗口。

Symbol Tree (符号树)

- 位于左侧，`Functions` 窗口旁边。
- 它以树状结构列出了程序中所有的符号，包括函数、标签、导入/导出函数等。你可以通过过滤器快速查找特定函数。

Data Type Manager (数据类型管理器)

- 左下角，`Functions` 窗口下方。
- 这里管理着程序中所有的数据类型 (struct, union, enum 等)。你可以创建、修改、导入和导出数据类型定义。这对于分析复杂的数据结构至关重要。

脚本化与自动化

Ghidra 强大的脚本能力是其核心优势之一。

打开 Script Manager

在 Code Browser 中，点击顶部菜单栏的绿色播放按钮图标，打开 Script Manager。

选择与运行脚本

这里有大量 NSA 官方和社区贡献的预置脚本，覆盖了从查找密码、解密数据到识别特定代码模式等各种任务。

编写自己的脚本

你可以通过 `Create New Script` 按钮创建新的 Java 或 Python 脚本。

Ghidra 提供了丰富的 API (称为 `FlatAPI`)，让你可以在脚本中访问和修改程序的几乎所有信息，例如：

```
# A simple Python script example that prints all function names and addresses
from ghidra.program.model.symbol import SymbolType

print("--- All Functions ---")
func_manager = currentProgram.getFunctionManager()
funcs = func_manager.getFunctions(True) # True means iterate in address order
for func in funcs:
    print("{} at {}".format(func.getName(), func.getEntryPoint()))
```

优缺点分析

优点

- 免费与开源：无任何费用，社区可以审查和贡献代码。

- 强大的反编译器: 内置的高质量反编译器是其最大的卖点, 足以媲美甚至在某些方面超越昂贵的商业软件。
- 跨平台: 基于 Java, 可以在 Windows, macOS, Linux 上无差别运行。
- 优秀的协作功能: Ghidra Server 的存在使得团队协作变得非常容易。

缺点

- 性能: 基于 Java Swing 的 UI 在处理超大型二进制文件时, 可能会感到卡顿, 性能不如 IDA Pro。
- 生态系统: 虽然正在快速发展, 但插件和社区支持的成熟度仍然不及 IDA Pro 经营多年的生态。
- 原生调试器: Ghidra 的调试器功能相对较弱, 不如 IDA Pro 和 x64dbg 等专用调试器成熟。

[T08] IDA Pro 入门指南

T08: IDA Pro 入门指南

IDA Pro (Interactive Disassembler Professional) 是由 Hex-Rays 公司开发的一款业界闻名的交互式反汇编器。在逆向工程领域，IDA Pro 被广泛认为是黄金标准，以其最强大的反汇编引擎、无与伦比的处理器支持和极其成熟的生态系统，成为专业人士进行软件分析、漏洞挖掘和恶意软件研究的首选工具。

目录

1. 核心特性
 2. IDA Pro vs. Ghidra vs. Radare2
 3. 版本与安装
 4. 基本工作流程
 5. 关键视图与快捷键
 6. 脚本与插件
 7. 优缺点分析
-

核心特性

顶级的反汇编引擎

IDA Pro 的核心竞争力在于其无与伦比的静态反汇编能力。它能够智能地、递归地遍历代码，区分代码与数据，识别函数边界，其分析结果的准确性是业界公认的最高水准。

FLIRT 技术

Fast Library Identification and Recognition Technology。这是 IDA 的标志性技术，通过对标准编译器库函数的签名进行模式匹配，能够自动识别并命名大量的库函数，极大地减少了逆向工程师的重复工作。

强大的交互性

IDA 的设计哲学鼓励用户与反汇编结果进行交互。用户可以随时重命名变量、修改类型、添加注释、转换数据格式，这些交互操作会实时地影响整个分析数据库。

Hex-Rays 反编译器

IDA Pro 的杀手级应用是其配套的 Hex-Rays 反编译器。虽然需要额外付费，但它被公认为目前市面上最强大的 C/C++ 反编译器，生成的伪代码质量极高，可读性极强。

多平台调试器

内置了强大的跨平台调试器，支持本地和远程调试，允许动态分析和修改程序行为。

极其丰富的插件生态

经过数十年的发展，IDA Pro 积累了海量的第三方插件，覆盖了从漏洞扫描、代码着色、数据解密到与其他工具联动的方方面面，极大地扩展了其功能边界。

IDA Pro vs. Ghidra vs. Radare2

特性	IDA Pro	Ghidra	Radare2
价格	非常昂贵	完全免费	完全免费
开源	否	是 (Java)	是 (C)
核心优势	最强的交互式反汇编	高质量的免费反编译器	极致的脚本化和命令行
UI	Qt, 业界标准, 成熟稳定	Java Swing, 功能强大但略显笨重	命令行, 或通过 Cutter 提供 GUI
反编译器	Hex-Rays (业界顶尖, 需付费)	内置免费, 质量非常高	内置免费 (ghidra-dec), 或支持其他
生态系统	极其成熟, 插件海量	快速发展中	高度可定制, 但插件较少
处理器支持	最广泛	广泛, 但略少于 IDA	极广, 覆盖很多小众架构

版本与安装

版本类型

版本	说明
IDA Pro	完整版本，包含所有处理器模块和调试器
IDA Home	针对个人爱好者的廉价版，功能受限
IDA Free	免费版本，功能严重受限，仅支持 x86/x64，且不能保存数据库

购买与安装

- 需要通过官方或授权经销商购买
- 安装过程是标准的下一步式安装
- 免费版仅适合非常初级的学习

基本工作流程

1. 启动 IDA

打开 IDA Pro。

2. 加载文件

在启动界面点击 `New`，或将二进制文件直接拖入主窗口。

3. 加载选项

IDA 会弹出一个加载对话框，让你确认文件类型、处理器类型等。通常，IDA 的自动分析非常准确，直接点击 `OK` 即可。

4. 自动分析

IDA 会进行长时间的自动分析。分析过程可以在底部的输出窗口看到。耐心等待分析完成是使用 IDA 的好习惯，否则很多功能无法正常使用。

5. 开始分析

分析完成后，即可开始交互式分析。

关键视图与快捷键

IDA View (反汇编视图)

这是 IDA 的主视图。按空格键可以在图形视图（流程图）和文本视图（线性反汇编）之间切换。

图形视图非常适合理解函数内的逻辑分支和循环。

Hex View (十六进制视图)

以经典的十六进制编辑器形式展示文件内容，与反汇编视图同步高亮。

Structures (结构体视图)

- 快捷键: `Shift+F9`
 - 用于定义和管理 C 语言风格的结构体
 - 你可以手动创建，也可以从 C 头文件导入
 - 正确地定义数据结构是逆向工程的关键一步
-

Enums (枚举视图)

- 快捷键: Shift+F10
- 用于定义和管理枚举类型
- 可以极大地提高代码的可读性, 例如将 mov eax, 2 变为 mov eax, MODE_READ

核心快捷键

快捷键	功能
G	跳转到指定地址
N	重命名变量、函数、标签
Y	修改变量类型
X	查看交叉引用 (cross-references), 即哪些地方调用/引用了当前符号
P	创建一个函数
U	取消定义 (如将代码变为未定义数据)
;	添加行注释
:	添加可重复注释
F5	(如果已购买) 启动 Hex-Rays 反编译器

脚本与插件

IDA 的强大能力有一半来自于其脚本和插件系统。

IDAPython

这是目前最主流的脚本语言。IDA 内置了一个完整的 Python 解释器和丰富的 API，允许你用 Python 脚本与 IDA 数据库进行深度交互。几乎所有重复性工作都可以通过 IDAPython 自动化。

```
# 示例: 打印所有函数名和地址
import idautools
import idc

for func_ea in idautools.Functions():
    func_name = idc.get_func_name(func_ea)
    print(f"{func_name} at {hex(func_ea)}")
```

IDC

IDA 自带的类 C 脚本语言。语法古老，功能不如 IDAPython 强大，但对于一些简单的任务仍然有用。

插件

IDA 的插件机制允许开发者使用 C++ 编写高性能插件，并将其深度集成到 IDA 的 UI 和核心中。社区中有大量优秀的开源插件，如 [FindCrypt](#)，[Keypatch](#) 等。

优缺点分析

优点

优点	说明
最强的反汇编质量	业界公认的、最可靠的静态分析结果
FLIRT 和类型系统	极大地自动化了库函数和数据结构的识别过程
成熟和稳定	经过多年打磨，软件本身极为稳定，用户体验流畅
强大的生态	海量的插件、教程和社区支持，遇到任何问题几乎都能找到解决方案
顶级的反编译器	Hex-Rays 反编译器是其最强大的护城河

缺点

缺点	说明
价格昂贵	对于个人开发者或小型团队来说，价格是最大的门槛
闭源	核心功能是黑盒，无法审查或修改
协作不便	原生不支持多人协作，需要依赖第三方插件

[T09] Radare2 入门指南

T09: Radare2 入门指南

Radare2 (通常简称为 r2) 是一款开源、免费、命令行驱动的逆向工程框架。它不仅仅是一个反汇编器，更像是一个功能极其丰富的“瑞士军刀”，集成了十六进制编辑、反汇编、调试、代码分析、漏洞利用、数据可视化等多种功能。Radare2 以其高度的可脚本化和可扩展性而闻名，深受寻求自动化和深度定制的黑客、CTF 选手和安全研究员的喜爱。

目录

1. 核心理念与特性
2. Radare2 vs. IDA Pro vs. Ghidra
3. 安装与入门
4. 基本命令与工作流程
5. Cutter - Radare2 的 GUI
6. 脚本化
7. 优缺点分析

核心理念与特性

命令行驱动

Radare2 的所有核心功能都通过命令行接口暴露。这使得它非常适合在终端、SSH 会话或脚本中运行，易于实现自动化。

模块化设计

其功能由一系列单字母命令和子命令构成，例如 `p` 用于打印 (print), `a` 用于分析 (analyze), `d` 用于调试 (debug)。这种设计遵循了 Unix 哲学。

海量架构支持

Radare2 支持数量惊人的处理器架构，包括许多非常小众和古老的嵌入式架构，这方面甚至超过了 IDA Pro。

高度可脚本化

你可以使用任何你喜欢的语言（Python, Go, JavaScript, Rust 等）通过 r2pipe 与 Radare2 实例进行交互，实现复杂的自动化分析流程。

内置调试器

集成了功能强大的多平台调试器，支持硬件断点、跟踪等高级功能。

强大的二进制文件解析

不仅支持 ELF, PE, Mach-O 等标准格式，还能解析文件系统、图片、文档等各种二进制 blob。

Radare2 vs. IDA Pro vs. Ghidra

特性	Radare2	IDA Pro	Ghidra
价格	完全免费	非常昂贵	完全免费
开源	是 (C)	否	是 (Java)
核心优势	极致的脚本化和命令行	最强的交互式反汇编	高质量的免费反编译器
UI	命令行 (或 Cutter GUI)	Qt, 业界标准, 成熟稳定	Java Swing, 功能强大但略显笨重
学习曲线	非常陡峭	中等	中等
自动化	设计哲学核心, 能力极强	主要通过 IDC/IDAPython	强大的 Headless 模式
灵活性	最高, 一切皆可定制	较低, 依赖插件	较高, 可通过插件扩展

安装与入门

安装

最推荐的安装方式是通过 `git` 克隆官方仓库并运行安装脚本：

```
git clone https://github.com/radareorg/radare2
cd radare2
sys/install.sh
```

启动

```
# 打开文件并分析  
r2 /bin/ls  
  
# 打开文件并进入调试模式  
r2 -d /bin/ls
```

基本命令与工作流程

Radare2 的命令结构是 [命令] [子命令] [参数]。例如 pdf 是 p (print) -> d (disassemble) -> f (function) 的组合，意为"打印函数反汇编"。

核心概念：万物皆 ?

在任何命令后面加上 ? 都可以查看该命令的帮助文档。这是学习 Radare2 最重要的方法。

命令	说明
?	显示顶级帮助
a?	显示分析 (analyze) 命令的帮助
pdf?	显示打印函数反汇编命令的帮助

分析 (a)

在你对一个二进制文件做任何事情之前，通常需要先分析它。

命令	说明
aaa	自动分析所有（函数、符号等）。这是最常用的起手命令
afl	列出所有已识别的函数 (Analyze Function List)
af	分析函数

打印 (p)

用于以不同格式显示数据。

命令	说明
px	以十六进制格式打印 (Print heXadecimal)
ps	打印字符串 (Print String)
pd N	反汇编 N 条指令 (Print Disassembly)
pdf	打印当前函数的反汇编 (Print Disassembly Function)

信息 (i)

用于显示文件的元信息。

命令	说明
ii	显示文件基本信息 (入口点、架构等)
is	显示符号
is	显示段 (sections)

Seek (s)

用于在文件中跳转。

命令	说明
s main	跳转到 main 函数的地址
s 0x8048400	跳转到指定地址
s-	撤销上一次跳转

可视化 (v)

Radare2 提供了强大的文本模式可视化功能。

操作	说明
v	进入可视化模式
q	退出可视化模式
p / P	在可视化模式下切换不同视图（反汇编、十六进制、调试寄存器等）
v	可视化模式下展示函数的 ASCII-art 流程图

Cutter - Radare2 的 GUI

对于不习惯纯命令行的用户，社区开发了 Cutter。Cutter 是一个基于 Qt C++ 的图形用户界面，后端由 Radare2 驱动。

主要特点

- 提供了类似 IDA Pro 和 Ghidra 的图形化界面
- 包括反汇编窗口、反编译窗口（集成了 Ghidra Decompiler）、函数列表、Hexdump 等
- 所有在 Cutter 中进行的操作，实际上都是在后台调用 Radare2 的命令完成的
- 对于初学者来说，从 Cutter 入手可以极大地降低学习 Radare2 的门槛

脚本化

Radare2 的精髓在于自动化。`r2pipe` 是其官方的脚本库，支持多种语言。

以下是一个 Python 脚本示例，用于打开一个文件，分析它，并打印所有函数的名称：

```
import r2pipe

# 打开文件
r2 = r2pipe.open("/bin/ls")

# 运行 'aaa' 命令进行分析
r2.cmd('aaa')

# 运行 'aflj' 命令获取 JSON 格式的函数列表并解析
functions = r2.cmdj('aflj')

# 打印每个函数名
if functions:
    for func in functions:
        print(f"Function found: {func['name']} at {hex(func['offset'])}")
```

优缺点分析

优点

优点	说明
无与伦比的脚本化能力	设计哲学使其成为自动化逆向分析的理想选择
极高的灵活性和定制性	你可以按照自己的需求组合命令，构建工作流
轻量与快速	核心程序非常小，运行速度快，资源占用少
海量架构支持	对各种奇异架构的支持是其一大特色
完全免费开源	无任何费用，社区可以审查和贡献代码

缺点

缺点	说明
陡峭的学习曲线	命令繁多，语法特殊，对新手非常不友好
文档相对混乱	虽然有帮助系统，但官方文档的结构性和完整性不如商业软件
默认反编译器	内置的反编译器质量不如 Ghidra 或 Hex-Rays，但可以通过插件集成 Ghidra Decompiler

速查手册

[T10] ADB 命令速查

T10: 常用 ADB 命令大全

ADB (Android Debug Bridge) 是一个功能强大的命令行工具，可让您与模拟器实例或连接的 Android 设备进行通信。

目录

- 常用 ADB 命令大全
 - 目录
 - 设备管理
 - 文件管理
 - 应用管理
 - 网络
 - 系统与调试
 - Logcat 日志查看
 - 高级 Shell 命令

设备管理

命令	描述
<code>adb devices -l</code>	列出所有连接的设备及其详细信息
<code>adb reboot</code>	重启设备
<code>adb reboot bootloader</code>	重启到引导加载程序 (Bootloader)
<code>adb reboot recovery</code>	重启到恢复模式 (Recovery)
<code>adb root</code>	以 root 权限重启 adbd 服务
<code>adb shell getprop ro.product.model</code>	获取设备型号
<code>adb shell getprop ro.build.version.release</code>	获取 Android 系统版本
<code>adb shell wm size</code>	获取屏幕分辨率
<code>adb shell wm density</code>	获取屏幕像素密度 (DPI)

文件管理

命令	描述
<code>adb push <本地路径> <远程路径></code>	将文件或文件夹从电脑推送到设备
<code>adb pull <远程路径> [本地路径]</code>	将文件或文件夹从设备拉取到电脑
<code>adb shell ls <路径></code>	列出设备指定路径下的文件和文件夹
<code>adb shell cd <路径></code>	切换设备上的当前目录
<code>adb shell pwd</code>	显示设备上的当前工作目录
<code>adb shell cp <源路径> <目标路径></code>	在设备上复制文件
<code>adb shell mv <源路径> <目标路径></code>	在设备上移动或重命名文件
<code>adb shell rm <文件路径></code>	在设备上删除文件
<code>adb shell mkdir <路径></code>	在设备上创建新目录

应用管理

命令	描述
<code>adb install <apk路径></code>	安装应用
<code>adb install -r <apk路径></code>	重新安装应用（保留数据）
<code>adb install -g <apk路径></code>	为应用授予所有运行时权限
<code>adb uninstall <包名></code>	卸载应用
<code>adb shell pm list packages</code>	列出所有已安装的应用包名
<code>adb shell pm list packages -f</code>	列出所有已安装的应用包名及其 APK 路径
<code>adb shell pm list packages -3</code>	列出所有第三方应用包名
<code>adb shell pm path <包名></code>	获取指定应用的 APK 路径
<code>adb shell am start -n <包名>/<Activity名></code>	启动一个 Activity
<code>adb shell am force-stop <包名></code>	强制停止应用
<code>adb shell pm clear <包名></code>	清除应用数据和缓存
<code>adb shell dumpsys activity grep mFocusedActivity</code>	获取当前前台 Activity

网络

命令	描述
<code>adb forward tcp:<PC端口> tcp:<设备端口></code>	将电脑端口的请求转发到设备端口
<code>adb forward --list</code>	列出所有端口转发规则
<code>adb forward --remove-all</code>	移除所有端口转发规则
<code>adb shell netstat</code>	查看网络状态 (监听的端口、连接等)
<code>adb shell ifconfig</code> or <code>adb shell ip addr</code>	查看网络接口信息和 IP 地址

系统与调试

命令	描述
<code>adb shell ps</code>	查看设备上的进程列表
<code>adb shell top</code>	查看实时资源占用情况
<code>adb shell dumpsys <服务名></code>	Dump 指定系统服务的信息 (如 <code>activity</code> , <code>battery</code> , <code>wifi</code>)
<code>adb shell screencap /sdcard/ screenshot.png</code>	截屏并保存到设备
<code>adb shell screenrecord /sdcard/ demo.mp4</code>	录制屏幕 (Ctrl+C 停止)
<code>adb bugreport [路径]</code>	生成并拉取完整的 bug 报告
<code>adb jdwp</code>	列出设备上可供调试的 Java 进程 ID (JDWP)

Logcat 日志查看

| 命令 | 描述 | | ----- | ----- | ----- | |
adb logcat | 实时打印设备日志 | | adb logcat -c | 清除旧的日志缓存 | | adb logcat -d |
Dump 当前日志到屏幕并退出 | | adb logcat -f /sdcard/log.txt | 将日志输出到设备上的文
件 | | | adb logcat *:S <标签>:<优先级> | 按标签和优先级过滤日志 | |
adb logcat | grep <关键词> | 在日志中搜索关键词 (区分大小写) |

- 日志优先级:

- `V` — Verbose (最低)

- `D` — Debug

- `I` — Info

- `W` — Warning

- `E` — Error

- `F` — Fatal

- `S` — Silent (最高)

- 示例: `adb logcat *:S MyApp:D` 只显示标签为 "MyApp" 且优先级为 Debug 或更高的日志。
-

高级 Shell 命令

命令	描述
<code>adb shell input text '<文本>'</code>	向当前输入框输入文本（不支持中文）
<code>adb shell input keyevent <按键码></code>	发送一个按键事件（例如 3 =HOME, 4 =BACK, 26 =POWER）
<code>adb shell input tap <x> <y></code>	模拟在屏幕指定坐标的单击事件
<code>adb shell input swipe <x1> <y1> <x2> <y2> [时长ms]</code>	模拟滑动事件
<code>adb shell settings get <命名空间> <键></code>	获取系统设置项的值
<code>adb shell settings put <命名空间> <键> <值></code>	修改系统设置项的值
<code>adb shell content query --uri <URI></code>	查询 Content Provider 中的数据
<code>adb shell ime list -s</code>	列出可用的输入法
<code>adb shell ime set <输入法ID></code>	设置默认输入法

案例研究

[C01] 反分析技术案例

C01: 反分析技术案例

为了保护其核心代码和数据不被轻易分析，现代 App 普遍采用了一系列的反分析技术。这些技术旨在检测和阻止调试器、Hook 框架（如 Frida）和模拟器的运行。本案例将分类介绍这些技术的实现原理和常见的绕过策略。

1. 反调试 (Anti-Debugging)

目标: 检测 App 是否正被调试器附加。

案例: 基于 `TracerPid` 的检测

这是最常见的一种反调试方法。在 Linux 内核中，每个进程的 `/proc/<pid>/status` 文件都记录了其状态信息，其中 `TracerPid` 字段表示正在追踪（调试）该进程的进程 PID。如果一个进程没有被调试，该值为 0。

实现原理: App 在运行时会启动一个独立的线程或子进程，周期性地读取自身的 `TracerPid`。

```
// Native (C/C++) implementation
#include <stdio.h>
#include <string.h>

int check_tracer_pid() {
FILE *fp = fopen("/proc/self/status", "r");
if (fp == NULL) {
return 0;
}

char line[128];
while (fgets(line, sizeof(line), fp)) {
if (strncmp(line, "TracerPid:", 10) == 0) {
int tracer_pid = 0;
sscanf(line, "TracerPid:\t%d", &tracer_pid);
fclose(fp);
return tracer_pid;
}
}
fclose(fp);
return 0;
}

// Call this in a loop somewhere in the App
if (check_tracer_pid() != 0) {
// Debugger detected, execute exit or crash logic
exit(0);
}
```

2. 反 Hook (Anti-Hooking)

目标: 检测和阻止 Frida 等 Hook 框架的注入和功能。

案例: 扫描内存中的 Frida 特征

Frida 在注入到目标进程后, 会在内存中留下一些特征, 如其核心库 `frida-agent.so`。

实现原理: App 会扫描自身的内存映射 (`/proc/self/maps`) , 寻找是否存在包含 `frida` 或 `gumjs` 等关键词的库。

```
// Native (C/C++) implementation
int check_for_frida_in_maps() {
FILE *fp = fopen("/proc/self/maps", "r");
if (fp == NULL) {
return 0;
}

char line[256];
while (fgets(line, sizeof(line), fp)) {
if (strstr(line, "frida-agent") || strstr(line, "gumjs")) {
fclose(fp);
return 1; // Frida detected
}
}
fclose(fp);
return 0;
}
```

3. 反模拟器 (Anti-Emulator)

目标: 检测 App 是否运行在模拟器 (如 Genymotion, Android SDK Emulator) 而非真实设备上。

案例: 检测设备特有文件或属性

模拟器通常会留下一些区别于真机的特有文件、驱动或系统属性。

实现原理: 检查系统属性: 通过 `getprop` 或直接读取 `build.prop` 文件, 检查是否存在 `ro.kernel.qemu`, `ro.hardware.goldfish` 等模拟器特有的属性。

检查文件: 检查是否存在 `/system/lib/libc_malloc_debug_qemu.so` 或 `/sys/qemu_trace` 等文件。

检查 CPU 信息: 读取 `/proc/cpuinfo`, 检查 `Hardware` 字段是否包含 `Goldfish` 或 `Intel` 等, 而非 `Qualcomm`, `MediaTek` 等移动端处理器厂商。

绕过策略: Hook `System.getProperty`: 在 Java 层 Hook 该方法, 当请求特定属性时返回一个伪造的、看起来像真机的值。

Hook 文件 API: Hook `File.exists()` 或 Native 层的 `access()`, `stat()` 等函数, 对特定的模拟器文件路径返回 `false`。

使用定制 ROM: 在一个修改过的 Android ROM 中, 可以从系统层面移除或伪造这些模拟器特征。

选择更逼真的模拟器: 一些商业或开源的、高度定制化的模拟器在隐藏自身特征方面做得更好, 更难被检测。

总结

反分析技术的攻防是一个不断升级的"猫鼠游戏"。

检测方: 努力寻找分析工具 (调试器、Frida) 在目标系统中留下的任何蛛丝马迹。

绕过方: 努力抹去或伪造这些痕迹, 让 App 认为自己运行在一个"干净"的环境中。

成功的绕过往往需要多项技术的组合, 从 Java 层的 Hook, 到 Native 层的 Patching, 再到对操作系统和工具链本身的定制。

[C02] 音乐 App 案例

C02: 音乐 App 案例

音乐类 App 是非常典型的逆向分析目标。其核心场景通常围绕着 VIP 会员特权、音频数据加密和客户端风控策略。本案例将模拟对一个典型音乐 App 的分析过程。

核心分析目标

1. 解锁 VIP 功能: 免费收听付费歌曲、下载无损音质、去除广告、使用专属皮肤等。
2. 音频数据提取: 分析加密的音频文件格式 (如 `ncm`, `qmcflac`) , 提取出可播放的 `mp3` 或 `flac` 文件。
3. API 分析: 分析其歌曲搜索、歌单获取、评论区等 API, 为第三方工具或爬虫提供支持。

案例：分析 VIP 歌曲的播放流程

第 1 步：定位切入点

目标: 找到判断用户是否为 VIP 以及歌曲是否为付费歌曲的关键代码。

1. 界面分析: 在 App 中播放一首需要 VIP 的歌曲, 通常会弹出一个"开通 VIP"的提示框。这个提示框是绝佳的切入点。
2. 寻找关键词: 使用 `jadx-gui` 反编译 APK, 搜索与弹窗内容相关的字符串, 例如"仅限 VIP"、"开通会员"等。
3. 交叉引用: 对找到的字符串进行交叉引用, 定位到显示这个弹窗的代码。你很可能会找到一个类似 `showVipDialog()` 的方法。
4. 回溯调用栈: 继续对 `showVipDialog()` 进行交叉引用, 向上回溯。通常, 你会找到一个包含了核心判断逻辑的函数, 其伪代码可能如下:

```
void onPlayButtonClick(Song song) {
    // isVip() determines from user information
    // song.isPaywalled() determines from song information
    if (!isVip() && song.isPaywalled()) {
        showVipDialog();
        return;
    }
    // ...execute playback logic...
    startPlayback(song);
}
```

目标: 绕过 VIP 判断, 让 App 认为我们是 VIP 用户。

最直接的方法是 Hook 负责判断用户身份的函数。

```
Java.perform(function () {
    // Assume UserInfo class manages user information
    var UserInfo = Java.use("com.example.music.model.UserInfo");

    // Directly hook isVip method to always return true
    UserInfo.isVip.implementation = function () {
        console.log("Bypassing VIP check, returning true.");
        return true;
    };

    // Some apps may also verify VIP expiration time
    UserInfo.getVipExpireTime.implementation = function () {
        // Return a timestamp far in the future
        return new Date(2099, 11, 31).getTime();
    };
});
```

- 请求的 URL 中带有 `quality=flac` 或 `hires` 等参数。
- 服务器返回的响应 `Content-Type` 可能不是 `audio/mpeg`, 而是一些自定义的类型如 `application/octet-stream`。

- 下载下来的文件（例如，`song.ncm`）无法用标准播放器播放。

1. 定位解密代码：这是最关键的一步。数据解密逻辑通常在 Native 层（`.so` 文件）以提高性能和逆向难度。
 - 关键词搜索：在 IDA Pro 或 Ghidra 中打开相关的 `.so` 文件，搜索 `aes`, `cbc`, `decrypt`, `RC4` 等加密算法相关的字符串。
 - JNI 入口：从 Java 层调用 Native 代码需要通过 JNI（Java Native Interface）。在 Java 代码中寻找 `native` 关关键字声明的函数，例如 `private native byte[] decryptAudio(byte[] encryptedData, int core);`。这个函数名就是你在 `.so` 文件中要找的符号。
 - Hook Native 函数：一旦定位到 JNI 函数（如 `Java_com_example_music_player_NativeDecoder_decryptAudio`），就可以使用 Frida 进行 Hook，观察其输入和输出。

```
Interceptor.attach(
    Module.findExportByName(
        "libaudiodecrypt.so",
        "Java_com_example_music_player_NativeDecoder_decryptAudio"
    ),
    {
        onEnter: function (args) {
            // args[0] is JNIEnv*, args[1] is jclass, args[2] is encrypted data jbyteArray
            console.log("Entering decryptAudio...");
            // Can save encrypted data for subsequent offline analysis
            this.encryptedBuffer = args[2];
        },
        onLeave: function (retval) {
            // retval is the decrypted jbyteArray
            console.log("Leaving decryptAudio. Decrypted data pointer: " + retval);
            // Here you can read the memory pointed to by retval to get the decrypted PCM or
            MP3 data
        },
    }
);
```

通过动态分析，你已经能够获取到解密后的音频数据。但如果想开发一个独立的、离线的格式转换工具，就需要彻底理解其加密方案。

- 静态分析 Native 代码：在 Ghidra/IDA 中仔细分析 `decryptAudio` 函数的逻辑。它可能包含：

-
- 元数据解析: 从加密文件头部读取歌曲 ID、专辑封面、比特率等信息。
 - 密钥派生: 使用一个固定的 Core Key 和从文件元数据中提取的 Nonce 来派生出每个文件唯一的 AES Key。
 - 解密循环: 循环读取加密的音频帧, 使用 AES 或其他算法进行解密。
 - 代码实现: 使用 Python 的 `cryptography` 等库, 将你在 Native 代码中看到的逻辑重新实现一遍。最终, 你就能开发出一个可以将 `.ncm` 批量转换为 `.flac` 的工具。
-

主流平台加密方案实例

虽然通用的分析思路是一致的, 但不同平台的具体实现细节各不相同。了解这些特征有助于更快地定位问题。

网某云音乐 (`.ncm`)

- 文件格式: `.ncm`
 - 加密细节: 采用 AES + RC4 的混合加密方案。
1. 元数据 (Meta): 文件中包含一块加密的元数据区域, 其中含有歌曲名、专辑封面、AES Key 等信息。这块区域本身使用一个固定的 Meta Key 进行 AES-ECB 解密。
 2. 音频数据 (Audio): 音频帧数据使用 AES-ECB 加密。解密所需的 AES Key 就存在于上一步解密后的元数据中。然而, 最终的解密密钥流是通过一个类似 RC4-KSA 的算法, 基于这个 AES Key 生成的。
- 逆向切入点:
 - 在 SO 库中搜索字符串 `ncm`, `core`, `meta`, `AES`, `RC4`。
 - 其解密逻辑通常被封装在一个或多个专门的 Native 函数中。

Q某音乐 (.qmcflac , .mflac , .qmc0)

- 文件格式: .qmcflac , .qmc0 , .qmc3 , .mflac 等。
- 加密细节: 未使用标准加密算法, 而是一套自定义的字节置乱 (Scramble) 方案。
- 其核心是依赖一个巨大的静态映射表 (Seed Map), 这个表硬编码在 SO 文件中。
- 解密时, 根据当前字节在文件中的偏移量, 通过一个复杂的公式计算出在映射表中的索引, 然后取出表中的值与加密字节进行运算 (通常是异或) 。
- 逆向切入点:
 - 由于没有使用标准算法, 搜索加密关键词是无效的。
 - 逆向的关键是在 SO 文件中找到那个巨大的静态数组 (映射表) 。
 - 定位一个紧凑的循环, 该循环体内部包含了复杂的偏移量计算和查表操作。

某狗音乐 (.kgm , .vpr)

- 文件格式: .kgm , .vpr 。
- 加密细节: 同样是自定义的置乱算法, 与 Q某音乐思路相似, 但实现不同。
- 依赖多个静态表 (通常在开源项目中被称为 table1 , table2) 。
- 文件头包含了解密所需的关键信息, 如密钥长度等。解密密钥由文件头信息和静态表共同派生。
- 逆向切入点:
 - 分析文件头的解析逻辑。
 - 定位多个静态表, 并还原其查表和密钥生成的算法。

某我音乐 (.kwm)

- 文件格式: .kwm 。

-
- 加密细节: 采用相对简单的 XOR 异或加密。
 - 解密密钥由一个硬编码在 SO 中的静态密钥 (Base Key) 与该歌曲的资源 ID (`rid`) 进行运算后得出。`rid` 是一个 `uin64_t` 类型的数字。
 - 得到最终密钥后, 对加密的音频数据进行逐字节异或即可完成解密。
 - 逆向切入点:
 - 搜索关键词 `rid`, `kwm`。
 - 定位一个逻辑相对简单的函数, 其包含了获取 `rid`、与静态密钥进行运算、然后循环异或的过程。
-

API 签名与加密实战

以下是基于真实项目的 API 加密实现分析, 展示了如何逆向还原各平台的请求签名逻辑。

网某云音乐 API 加密实现

网某云音乐采用 AES + RSA 双层加密方案, Web 端和 App 端使用不同的加密策略。

核心密钥常量

```
class Music163:
    def __init__(self):
        # Web 端 AES 密钥和 IV
        self.aes_key = '0CoJUm6Qyw8W*****'
        self.aes_iv = '01020304050***08'

        # RSA 公钥参数
        self.rsa_exponent = '010001'
        self.rsa_modulus =
'00e0b509f6259df8642dbc35662901477df22677ec152b5ff68ace615bb7b725152b3ab17a876aea8a5aa
76d2e417629ec4ee341f56135fccf695280104e0312ecbda92557c93870114af6c9d05c4f7f0c3685b7a46
bee255932575cce10b424d813cfe4875d3e82047b97ddef52741d546b8e289dc6935b3ece0462db0a22***
*'

        # 随机字符集, 用于生成随机密钥
        self.words = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
```

Web 端加密流程 (params + encSecKey)

```
def aes(self, data, key, iv):
    """AES-CBC 加密, PKCS7 填充"""
    bs = AES.block_size
    pkcs7 = lambda t: t + (bs - len(t) % bs) * chr(bs - len(t) % bs)
    cryptor = AES.new(key, AES.MODE_CBC, iv)
    encrypt_text = cryptor.encrypt(str.encode(pkcs7(data)))
    return str(base64.encodebytes(encrypt_text), encoding='utf-8').strip()

def rsa(self, data, exponent, modulus):
    """RSA 加密 (无填充, 反向字节序) """
    res = int(codecs.encode(data[::-1].encode('utf-8'), 'hex_codec'), 16) ** int(exponent, 16) % int(modulus, 16)
    return format(res, 'x').zfill(256)

def get_formdata(self, data):
    """生成 Web 端请求参数"""
    # 1. 生成 16 位随机密钥
    random_key = ''.join([random.choice(self.words) for _ in range(16)])

    # 2. 两次 AES 加密: 先用固定 key, 再用随机 key
    params = self.aes(self.aes(data, self.aes_key, self.aes_iv), random_key,
                      self.aes_iv)

    # 3. RSA 加密随机密钥
    seckey = self.rsa(random_key, self.rsa_exponent, self.rsa_modulus)

    return {"params": params, "encSecKey": seckey}
```

App 端 eapi 接口加密

```
def md5_app_sign(self, path, data):
    """App 端签名算法"""
    string = "nobody" + path + "use" + data + "md5for*****"
    return hashlib.new("md5", string.encode()).hexdigest()

def encrypt_app_params(self, path, data):
    """App 端参数加密"""
    sign = self.md5_app_sign(path, data)
    # 使用固定分隔符拼接
    string = str.encode(path + "-36cd479****-" + data + "-36cd479****-" + sign)
    # PKCS7 填充
    string = string + (chr((16 - (len(string) % 16))).encode() * (16 - (len(string) % 16)))
    # AES-ECB 加密
    encryptor = AES.new(b"e82ckenh8dic****", AES.MODE_ECB)
    ciphertext = encryptor.encrypt(string)
    return binascii.b2a_hex(ciphertext).upper()

def decrypt_app_data(self, data):
    """App 端响应解密"""
    encryptor = AES.new(b"e82ckenh8dic****", AES.MODE_ECB)
    data = encryptor.decrypt(data).decode()
    return data[:data.rfind("}") + 1]
```

设备指纹 Cookie 构造

```
def get_app_cookie(self, device):
    """构造 App 端设备指纹 Cookie"""
    cookies = {
        "EVNSM": "1.0.0",
        "osver": device.get("os_version"),           # 系统版本
        "deviceID": device.get("device_id"),          # 设备 ID
        "appver": "9.1.0",                           # App 版本
        "NMDI": device.get("nmdi"),                  # 网易设备标识
        "NMCID": device.get("cid"),                  # 渠道 ID
        "versioncode": "9001000",
        "mobilename": device.get("model").replace(" ", ""), # 手机型号
        "resolution": device.get("resolution"),       # 分辨率
        "os": "android",
        "channel": "ali"                            # 渠道来源
    }

    if "music_a" in device:
        cookies["MUSIC_A"] = device.get("music_a")  # 登录凭证
    if "nmtid" in device:
        cookies["NMTID"] = device.get("nmtid")      # 跟踪 ID
    if "csrf" in device:
        cookies["__csrf"] = device.get("csrf")       # CSRF Token

    return "; ".join([f"{k}={v}" for k, v in cookies.items()])
```

实际请求示例

```
def search(self, keyword, offset):
    """歌手搜索 API 调用示例"""
    # 1. 构造请求数据
    data = json.dumps({
        "sub": "false",
        "s": keyword,
        "q_scene": "normal",
        "offset": str(offset),
        "queryCorrect": "true",
        "checkToken": self.random_check_token(), # 随机校验 Token
        "limit": "100",
        "header": "{}",
        "e_r": "true"
    }, ensure_ascii=False)

    # 2. 加密参数
    url = "https://interface.music.163.com/eapi/v1/search/artist/get"
    encrypted = self.encrypt_app_params("/api/v1/search/artist/get", data)

    # 3. 发送请求
    headers = self.get_app_headers()
    res = requests.post(url, data=f"params={encrypted.decode()}", headers=headers)

    # 4. 解密响应
    return json.loads(self.decrypt_app_data(res.content))
```

全某 K 歌/某狗音乐音频解密

全某 K 歌使用 `.tkm` 格式存储加密音频，解密算法基于预计算的异或表。

异或映射表生成

```
# 核心 Seed 映射表 (256 字节)
SEED_MAP = [
    0x77, 0x48, 0x32, 0x73, 0xDE, 0xF2, 0xC0, 0xC8, 0x95,
    0xEC, 0x30, 0xB2, 0x51, 0xC3, 0xE1, 0xA0, 0x9E, 0xE6,
    0x9D, 0xCF, 0xFA, 0x7F, 0x14, 0xD1, 0xCE, 0xB8, 0xDC,
    # ... 共 256 个字节
    0x4A, 0x11
]

class Mask:
    """生成解密掩码序列"""
    def __init__(self):
        self.index = 0

    def next(self):
        """计算下一个掩码字节"""
        v11 = self.index
        if v11 >= 0x8000:
            v11 %= 0xFFFF

        # 核心算法: 平方加常数取模后查表
        result = SEED_MAP[(v11 * v11 + 80923) % 256]
        self.index += 1
        return result
```

预计算异或文件生成

```
def generate_xbytes_file():
    """生成约 200MB 的异或表文件 (一次性生成)"""
    mask = Mask()
    with open("xbytes", "wb") as f:
        # 生成 209,771,520 字节的异或表
        # 这个大小足以覆盖大多数音频文件
        m = [mask.next() for _ in range(209771520)]
        f.write(bytes(m))
```

音频解密实现

```
import sys

class Kg:
    def __init__(self):
        self.xbytes_file = "/path/to/xbytes" # 预计算的异或表文件

    def tkm2m4a(self, tkm_data):
        """将加密的 .tkm 文件解密为 .m4a"""
        if len(tkm_data) > 209771520:
            return None # 文件过大, 超出异或表范围

        with open(self.xbytes_file, "rb") as xbytes_file:
            xbytes = xbytes_file.read()

        # 将字节序列转换为大整数进行异或运算
        int_tkm = int.from_bytes(tkm_data, sys.byteorder)
        int_xbytes = int.from_bytes(xbytes[:len(tkm_data)], sys.byteorder)

        # 核心解密: 整数异或
        m4a = (int_tkm ^ int_xbytes).to_bytes(len(tkm_data), sys.byteorder)
        return bytes(m4a)
```

伴奏下载完整流程

```
def download_accompany(self, mid):
    """下载并解密全民K歌伴奏"""
    # 1. 获取 vkey (访问凭证)
    vkey = self.get_vkey()

    # 2. 构造媒体 URL
    media_url = f"http://bsy.tsmusic.kg.qq.com/{media_mid}.tkm?vkey={vkey}&guid=1736440468&fromtag=0"

    # 3. 下载加密音频
    res = requests.get(media_url, headers=self.get_headers())
    encrypted_data = res.content

    # 4. 解密为 m4a
    decrypted = self.tkm2m4a(encrypted_data)

    # 5. 保存文件
    with open(f"{mid}.m4a", "wb") as f:
        f.write(decrypted)
```

某米音乐 API 签名

某米音乐使用 Token + MD5 签名机制。

```
class XiaMi:  
    def get_token_from_cookies(self, cookies):  
        """从 Cookie 中提取 Token"""  
        if cookies:  
            token = re.findall("xm_sg_tk=(.*?)_.*?;", cookies)  
            return token[0] if token else None  
  
    def get_sign(self, key, token, query=""):  
        """计算请求签名"""  
        # 获取 API 路径  
        path = self.get_path(self.urls.get(key))  
        # 签名公式: md5(token + "_xmMain_" + path + "_" + query)  
        return hashlib.md5(f"{token}_xmMain_{path}_{query}".encode()).hexdigest()  
  
    def search_songs(self, keyword, page=1):  
        """歌曲搜索示例"""  
        # 1. 获取 Token 和 Cookie  
        ua_token, cookies = self.get_cookies()  
        token = self.get_token_from_cookies(cookies)  
  
        # 2. 构造查询参数  
        query = json.dumps({"key": keyword, "pagingVO": {"page": page, "pageSize":  
            30}})  
  
        # 3. 计算签名  
        sign = self.get_sign("song_search", token, query)  
  
        # 4. 构造 URL  
        q = base64.b64encode(query.encode()).decode()  
        url = f"https://www.xiami.com/api/search/searchSongs?_q={q}&_s={sign}"  
  
        # 5. 发送请求  
        headers = self.get_headers(xmua=ua_token, cookies=cookies)  
        return requests.get(url, headers=headers).json()
```

某咕音乐 API 分析

某咕音乐的 API 相对简单，主要依赖 User-Agent 区分设备类型。

```
class Migu:  
    def __init__(self):  
        self.urls = {  
            "digital_album_list": "https://c.musicapp.migu.cn/MIGUM2.0/resource/  
dalbum/list-by-tag/v1.0?tagIds=&page={page}",  
            "digital_album_info": "https://c.musicapp.migu.cn/MIGUM2.0/v1.0/content/  
resourceinfo.do?needSimple=01&resourceId={resource_id}&resourceType=5",  
            "songlist_info": "https://c.musicapp.migu.cn/MIGUM3.0/resource/playlist/  
v2.0?playlistId={songlist_id}",  
            "songlist_songs": "https://c.musicapp.migu.cn/MIGUM3.0/resource/playlist/  
song/v2.0?playlistId={songlist_id}&pageNo={page}&pageSize=50"  
        }  
  
    def web_headers(self):  
        return {"user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) ..."}  
  
    def phone_headers(self):  
        return {"user-agent": "Mozilla/5.0 (Linux; Android 6.0; Nexus 5 Build/  
MRA58N) ..."}  
  
    def get_songlist(self, songlist_id, page=1):  
        """获取歌单详情"""  
        url = self.get_url("songlist_songs", songlist_id=songlist_id, page=page)  
        return requests.get(url, headers=self.phone_headers()).json()
```

逆向要点总结

密钥提取策略

平台	加密方式	密钥位置	提取难度
网某云音乐	AES+RSA	硬编码在 JS/Java 代码中	中
Q某音乐	静态映射表	硬编码在 SO 文件中	高
某狗音乐	多表异或	SO 文件 + 服务端	高
某我音乐	XOR + RID	SO 文件中静态密钥	低
某米音乐	MD5 签名	Cookie 中的 Token	低
某咕音乐	无加密	N/A	无

通用逆向流程

1. 抓包分析: 使用 Charles/Fiddler 捕获 HTTPS 请求, 识别加密字段
2. 定位加密点: 在 APK 中搜索 URL 路径或参数名, 定位加密函数
3. 分析算法: 静态分析 Java/Smali 代码, 动态 Hook 验证
4. 提取密钥: 从代码或内存中提取硬编码的密钥常量
5. 复现实现: 使用 Python 等语言还原加密逻辑
6. 绕过检测: 处理设备指纹、请求频率等风控策略

总结

这个案例展示了从客户端功能绕过, 到网络协议分析, 再到核心加密算法逆向的完整流程。它结合了 Java 层的 Hook 和 Native 层的分析, 是移动端逆向中非常具有代表性的场景。

通过分析真实的音乐平台爬虫项目，我们可以看到：

1. 加密复杂度差异：不同平台的加密强度差异明显，从简单的 MD5 签名到复杂的双层 AES+RSA
2. 音频加密特点：音频数据加密通常采用流式加密（XOR/RC4）以保证性能
3. 设备指纹重要性：现代 App 大量依赖设备指纹进行风控，需要完整模拟
4. Native 层保护：核心加密逻辑往往放在 SO 文件中，增加逆向难度

[C03] 社交媒体与风控案例

C03: 社交媒体 App 与风控案例

社交媒体 App（如 X、Instagram、TikTok）是爬虫和自动化工具最常光顾的地方。因此，这些 App 的开发者在客户端和服务器端都部署了极其复杂的安全与风控系统，以保护用户数据和平台生态。本案例将聚焦于这些 App 中常见的风控对抗技术。

核心分析目标

1. API 签名算法逆向: 几乎所有社交 App 的核心 API 请求都包含一个或多个签名参数（如 `X-Signature`, `X-Gorgon`）。这些签名是请求合法性的证明，也是逆向的主要目标。
2. 设备指纹分析: 理解 App 如何收集设备信息（硬件、软件、网络环境等）来生成唯一的设备 ID (`device_id`)，并用于风控决策。
3. 业务风控逻辑分析: 分析 App 在关键业务点（如注册、登录、点赞、评论）的风控策略，例如人机验证（滑块验证码、点图验证等）。

案例：分析一个典型社交 App 的 API 签名流程

第 1 步：定位签名参数

目标: 识别出 API 请求中哪个参数是签名。

1. 网络抓包: 使用 Charles 或 Mitmproxy 拦截 App 的网络流量。刷新首页动态 (timeline) 的请求是最好的分析对象，因为它通常包含了最复杂的签名。

2. 观察请求: 查看一个典型的 API 请求, 例如 `/api/v2/feed`。你会注意到其 URL 参数或请求头 (Headers) 中存在一些看起来像哈希值的、无明显语义的参数。

- URL 参数: `...&mas=01&as=a1...&ts=166...&ssmix=a...`

- 请求头: `X-Gorgon: 0404...`, `X-Khronos: 166...`

3. 参数筛选: 通过多次重复请求, 比较参数的变化规律。

- 不变的: `device_id`, `os_version` 等, 通常是设备指纹的一部分。

- 随时间变化的: `ts`, `X-Khronos` 等, 通常是时间戳。

- 每次请求都随机变化的: `mas`, `as`, `X-Gorgon` 等, 这些就是我们要找的核心签名。

第 2 步：定位签名生成代码

目标: 找到在客户端生成这些签名的代码。这是整个流程中最关键、也最困难的一步。

1. 全局搜索: 在 `jadgui` 中, 全局搜索上一步识别出的参数名, 如 `X-Gorgon`。如果运气好, 你能直接定位到构建网络请求的地方。

2. Hook 大法: 如果搜索无果 (通常是因为参数名在代码中被加密或混淆了), Frida Hook 将是你的主力武器。

- Hook 网络库: 从网络请求的源头入手。Hook `OkHttp` 的 `Request.Builder.addHeader` 或 `url()` 方法, 打印出调用栈。

```
Java.perform(function () {
    var Builder = Java.use("okhttp3.Request$Builder");
    Builder.addHeader.implementation = function (name, value) {
        if (name === "X-Gorgon") {
            // Found it! Print call stack
            console.log("Found X-Gorgon being added: " + value);
            console.log(
                Java.use("android.util.Log").getStackTraceString(
                    Java.use("java.lang.Exception").$new()
                )
            );
        }
        return this.addHeader(name, value);
    };
});
```

3. 静态分析签名函数: 定位到具体的签名函数后 (例如, `SignHelper.getSign(params)`) , 在 Ghidra 或 IDA 中仔细分析其逻辑。

◦ 输入: 它的输入通常是一个 `Map` 或 `List`, 包含了所有要参与签名的业务参数 (如 `user_id`) 和设备指纹参数。

◦ 逻辑: 函数内部逻辑通常是:

1. 对所有参数按 key 进行字典序排序。
2. 将排序后的 key-value 对拼接成一个长的字符串。
3. 将固定的盐 (salt, 可能硬编码或从 Native 获取) 拼接到字符串的头部或尾部。
4. 对最终的字符串进行 MD5 或 HMAC-SHA256 哈希。
5. 有时还会进行额外的变换, 如 Base64 编码或自定义的字节操作。

◦ Native 混淆: 越来越多的 App 将核心的签名算法 (特别是盐) 放到 `.so` 文件中, 并使用 OLLVM 或 VMP 等技术进行混淆, 以对抗静态分析。这时, 就需要结合动态调试来一步步跟踪其执行流程。

第 3 步：模拟签名与自动化

目标: 在你自己的 Python 或其他语言脚本中, 重新实现签名算法, 从而可以脱离 App 独立发起合法的 API 请求。

1. 代码复现: 根据静态分析的结果, 用 Python 完整地复现整个签名流程。每一个细节都要精确匹配, 包括参数的排序、拼接方式、哈希算法等。

2. 获取设备参数: 签名依赖的设备指纹参数 (`device_id`, `install_id` 等) 通常在 App 首次启动时生成并存储在本地。你需要 Hook 相关的函数来获取一套合法的设备参数, 并在你的脚本中使用它们。

3. 风控对抗:

◦ 滑块验证码: 当服务器检测到你的请求异常时 (例如, IP 地址异常、请求频率过高), 它可能会返回一个需要进行人机验证的响应。你需要分析验证码的逻辑, 这通常涉及到对一个 `JavaScript` 文件的逆向, 分析其滑块轨迹加密算法。

◦ 请求频率: 模拟真实用户的行为, 在请求之间加入随机的延迟。

-
- 代理 IP: 使用高质量的代理 IP 池来避免单个 IP 被封禁。
-

总结

社交媒体 App 的逆向是典型的"数据在客户端, 但由服务器规则校验"的场景。其核心是对抗, 而不只是解密。

- 签名是核心: 逆向签名算法是所有工作的基础。
- 动静结合: 需要反复在静态分析 (Ghidra/IDA) 和动态验证 (Frida) 之间切换。
- 风控是持续的斗争: 即使你成功逆向了签名, 服务器端的风控策略也在不断演进。这是一个长期的、动态的攻防过程。

[C04] App 加密签名案例

C04: 主流 App 加密签名机制案例

对主流 App 的 API 加密机制进行逆向分析，是检验和应用逆向工程综合能力的最佳实战。本案例研究将以常见的电商和社交类 App 为例，剖析其网络请求中核心加密字段和签名的生成逻辑，展示理论知识在实战中的应用。

免责声明：本文内容基于公开技术和过往分析经验的总结，旨在技术交流与学习。具体的加密实现会频繁更新，本文不保证与线上最新版本完全一致。

目录

- 通用加密与签名模式
- 案例分析 1：电商 App (类拼多多模式)
- 案例分析 2：社交 App (类某红书模式)
- 案例分析 3：某节跳动系 (某音/TikTok)
- 案例分析 4：某手
- 案例分析 5：某团
- 案例分析 6：某里系 (某宝、某付宝)
- 逆向分析通用策略
- 高级策略：黑盒 RPC 调用详解

通用加密与签名模式

在分析具体案例前，我们先了解几种行业内通用的 API 保护模式：

- 请求体加密：对整个 POST Body 进行对称（AES）或非对称（RSA）加密，保护数据内容隐私。
- 参数级加密：仅对请求中的个别敏感字段（如密码、手机号）进行加密。
- **sign** 签名机制：最核心、最普遍的模式。通过对请求参数、时间戳、随机数等进行组合和哈希，生成一个签名值。服务器端会以同样的方式计算签名并进行比对，用于：
 - 防篡改：确保传输过程中的数据未被修改。
 - 防重放：通过加入时间戳或 Nonce，让签名一次有效。
 - 身份认证：验证请求是否由合法的客户端发出。

案例分析 1：电商 App (类拼多多模式)

核心风控字段 (`anti_content`)

- 现象：在其 API 请求中，经常能看到一个名为 `anti_content` 的、内容极长的加密字段。
- 本质：它并非简单的 API 参数签名，而是一个由客户端 SDK 生成的、高度复杂的风控数据包。它更侧重于识别“人”与“机器”，而非认证 API 调用本身。
- 可能包含的内容：
 - 设备指纹：包含之前文档中提到的几乎所有硬件、软件和系统特征。
 - 环境检测：是否 Root、是否越狱、是否使用了 Hook 框架（Frida/Xposed）、是否在模拟器中运行。
 - 传感器数据：在特定时间段内采集的加速度计、陀螺仪数据，用于判断设备是否在正常物理状态下。
 - 行为数据：用户的点击坐标、滑动轨迹等。

-
- 逆向挑战: `anti_content` 的生成逻辑通常被封装在高度混淆的原生 SO 库中, 并可能包含 `SVC` 系统调用等反分析技术。完整复现其算法的难度极高。

API 认证签名 (`sign`)

- 现象: 除了 `anti_content`, 请求参数中还有一个相对独立的 `sign` 字段。
- 目的: 这个字段才是真正用于 API 级别认证的签名。
- 典型生成逻辑:
 - 收集所有请求参数 (GET Query Params 和 POST Form Body Params)。
 - 剔除 `sign` 字段本身。
 - 按参数名的 ASCII 字母顺序进行排序。
 - 将排序后的参数拼接成 `key=value&...` 的字符串 (空值参数可能不参与拼接)。
 - 在拼接好的字符串前后或中间插入一个固定的密钥 (App Secret / Salt), 这个密钥通常硬编码在 SO 文件中。
 - 对最终的字符串进行 MD5 或 HMAC-SHA256 哈希, 得到签名值。

案例分析 2: 社交 App (类某红书模式)

复杂的请求头签名 (`X-Sign`)

- 现象: 认证信息不放在 URL 参数中, 而是位于 HTTP 请求头, 如 `X-Sign`, `X-T` (时间戳),
`X-B3-TraceId` 等。
- `X-Sign` 的构成:
 - 格式: 通常是 `MD5(some_string)` 的形式。
 - `some_string` 的拼接方式: `URL Path + Sorted Query Params + (POST Body Hash) + Token/Salt`。
 - 这意味着, 不仅 URL 参数, 连 POST 的内容也参与了签名计算。

-
- 有时还会包含其他请求头的值。
 - 动态 Salt: 其签名用的密钥可能不是固定的, 而是部分由服务器下发, 或与时间戳、设备信息动态生成, 这使得暴力破解和简单模拟请求变得非常困难。

设备信息上报 (`X-DeviceInfo`)

- 现象: 有一个专门的请求头, 如 `X-DeviceInfo`, 其内容是加密或 Base64 编码后的 JSON 字符串, 里面是详细的设备指纹信息。
- 关联性: 服务端的风控系统会将 `X-Sign` 和 `X-DeviceInfo` 进行强关联校验。
 - 首先验证 `X-Sign` 是否合法。
 - 然后解码 `X-DeviceInfo`, 分析设备是否可信。
 - 最后, 可能会交叉验证, 例如, 某个版本的 App 是否可能运行在某个特定的 Android SDK 版本上, 如果不匹配, 则判定为异常。

案例分析 3: 某节跳动系 (某音/TikTok)

- 现象: 其 API 请求中包含多个复杂的自定义请求头, 如 `X-Gorgon`, `X-Khronos`, `X-Argus`, `X-Ladon`。请求体通常是经过 Protobuf 序列化后再加密的二进制数据。
- 核心逻辑:
 - 设备注册: App 首次启动时会进行设备注册 (`/service/2/device_register/`), 获取服务器下发的 `device_id` 和 `install_id`。这两个 ID 是后续所有业务请求的身份基础。
 - 多重签名系统: `X-Gorgon` 是最核心的 API 请求签名, 它将 URL、Cookie、POST Body 的哈希、设备指纹信息等多种因素混合计算而成。`X-Khronos` 是加密过的时间戳。这套体系确保了请求的来源、时效和完整性都可被验证。
 - Protobuf 序列化: 大量使用 Protobuf 进行数据交换, 相比 JSON, 它更高效, 但也增加了逆向难度, 因为分析者需要先找到或还原 `.proto` 文件才能理解数据结构。

-
- Cronet 网络库: 使用 Google 的 Cronet 网络库进行网络请求, 这使得常规的 OkHttp Hook 方法失效, 必须深入到更底层的 `cronet.so` 或系统网络调用层面去进行 Hook。
 - 逆向挑战:
 - 虚拟机保护 (VMP): 其核心 SO 库 (如 `libmetasec_ml.so`, `libmsaoaidsec.so`) 使用了行业顶级的 VMP 或其自研的虚拟机保护技术。这会将原始的 ARM 指令转换成虚拟机自定义的字节码, 导致 IDA 等工具无法进行静态分析。
 - 算法快速迭代: 签名算法几乎每个版本都在变化, 增加了长期维护的难度。
 - 分析策略: 鉴于 VMP 的存在, 完全还原签名算法几乎是不可能的。业界主流的策略是放弃算法还原, 转向 RPC 调用。即通过 Frida 等工具找到 SO 中负责计算签名的函数 (无论是导出还是非导出函数), 模拟其运行环境和参数, 直接调用它来获取签名结果。

案例分析 4: 某手

- 现象: API 请求参数中包含 `sig` 和 `_NS_sig` 字段。请求体同样可能使用 Protobuf 序列化并加密。
- 核心逻辑:
 - 双签名体系:
 - `sig`: 一个相对传统的 API 签名, 通常是对所有请求参数进行排序、拼接、加盐后进行 MD5 或 HMAC 哈希。
 - `_NS_sig` (New Signature): 这是一个更复杂的风控签名, 其计算过程融入了大量的设备指纹信息, 用于对抗模拟器和脚本。
 - 动态 Salt: 在加密和签名过程中会使用到一个 `client_salt`, 这个盐值并非固定, 而是可能从 Protobuf 数据中动态获取, 或者通过 JNI 调用 SO 库动态生成, 这增加了模拟请求的难度。

- 逆向挑战:

- 其核心 SO 库（如 `libcore.so`）经过了深度混淆，虽然可能不是 VMP 级别，但静态分析依然困难重重。
- 同样大量使用了 Protobuf，需要投入精力去逆向其数据结构。

案例分析 5：某团

- 现象：API 请求中包含大量自定义请求头，如 `M-TraceId`。请求体被加密，并且能看到 `rohr` 和 `mtgsig` 等新一代的风控及签名字段。

- 核心逻辑：

- 中心化风控库：核心保护逻辑高度集中在 `libmtguard.so` 中，该库负责生成几乎所有的签名和风控数据。
- 请求压缩与加密：请求体可能会先用 `zlib` 或 `gzip` 进行压缩，然后再通过 AES 进行加密，服务器端需要先解密再解压。
- `rohr` & `mtgsig`：这是其新一代的风控签名体系。`rohr` 是一个风控令牌，包含了加密的设备和环境信息；`mtgsig` 是 API 签名，它在计算时会依赖 `rohr` 的部分数据，两者强关联。
- 统一请求网关：有一个统一的 API 网关，加密和签名逻辑相对集中，便于统一管理和迭代。

- 逆向挑战：

- `libmtguard.so` 是逆向的重中之重，其内部逻辑复杂且经过混淆。
- 风控维度极广，除了常规的设备指纹，还可能包括地理位置、历史行为、网络环境等，对伪造设备画像的一致性要求非常高。

案例分析 6：某里系（某宝、某付宝）

- 现象：API 请求中包含一个 `sign` 字段，并且还有一个名为 `wua` 的神秘参数。网络请求通过自有的 MTop 网关进行分发。
- 核心逻辑：
 - 统一网关 (MTop)：某里系 App 使用自研的 MTop 作为统一无线网关。所有的 API 请求都经过这个网关，便于统一进行签名校验、安全风控和流量调度。
 - 安全核心 (`libsgmain.so`)：所有的安全逻辑都高度集成在 `libsgmain.so` 以及一系列 `libsgxxx.so` (如 `libsgsecuritybody.so`) 的安全组件中。这是某里安全的核心技术结晶，负责签名 `sign` 和风控参数 `wua` 的生成。
 - `sign` 签名：签名算法极其复杂。它不仅包含 API 的业务参数，还会将时间戳、App 版本、Token 以及从安全 SDK 中获取的大量设备指纹信息一同参与计算。其拼接和加密方式非常规整。
 - `wua` 风控参数：这是一个类似于 `anti_content` 的黑盒风控参数。它由 `libsgmain.so` 采集海量的设备信息（包括硬件、系统、网络、传感器、环境检测等）后，经过高度混淆的算法加密生成。`wua` 的生成难度和重要性甚至高于 `sign`。服务端会将 `sign` 和 `wua` 进行强关联校验。
 - ACCS 通道：使用自研的 ACCS (Alibaba Cloud Channel Service) 长连接通道，基于 HTTP/2，进一步封装了网络请求，使得常规抓包和分析变得更加困难。
- 逆向挑战：
 - 顶级混淆：`libsgmain.so` 及其依赖库使用了自研的、多层次的复杂混淆技术，静态分析几乎无法下手，是业界公认的最难逆向的 SO 库之一。
 - 动态加载与反调试：安全组件的加载和初始化过程非常隐晦，并伴有大量的反调试和环境检测手段，给动态调试和分析设置了极高的门槛。
 - 黑盒 RPC 调用：与某节系类似，业界的主流策略是放弃算法还原。逆向的终极目标是在 SO 文件中找到一个类似 `main` 的函数入口，通过 RPC 调用的方式，传入请求参数，获取计算好的 `sign` 和 `wua` 值。定位这个入口需要极其深厚的动态调试和二进制分析功底。

逆向分析通用策略

1. 静态分析 (Jadx/Ghidra):

- 全局搜索: 搜索关键词, 如 `sign`, `encrypt`, 以及上述案例中的 `anti_content`, `X-Sign`, `X-Gorgon`, `mtgsig`, `wua` 等。
- 定位网络库: 现代 App 大多使用 OkHttp。搜索 `okhttp3.Interceptor` 的实现类, 因为加密和签名的逻辑常常在自定义拦截器中统一处理。
- 追踪 JNI 调用: 找到 Java 层调用 Native 方法的地方, 重点关注那些函数名可疑 (如 `getSignFromC`)、参数多且包含字节数组的函数。

2. 动态分析 (Frida):

- Hook 加密算法: 这是最有效的方法。Hook `java.security.MessageDigest.digest` 和 `javax.crypto.Mac.doFinal`, 打印它们的输入 (即被签名的明文) 和调用堆栈, 可以瞬间定位到生成签名的代码位置。
- Hook 网络请求: Hook `okhttp3.Request.Builder` 的 `build()` 方法, 或 `okhttp3.OkHttpClient` 的 `newCall` 方法, 可以 dump 出所有即将发出的网络请求的完整信息 (URL, Headers, Body), 用于和抓包结果对比。
- Hook SO 函数: 定位到核心 SO 库后, 用 Frida `Interceptor.attach` 直接 Hook 目标导出函数, 观察其输入和输出。对于非导出函数, 可以通过基址加偏移的方式进行 Hook。

高级策略：黑盒 RPC 调用详解

在分析某节、某里等顶级厂商的加固 SO 时, 会发现其核心逻辑受 VMP (虚拟机保护) 或自研虚拟机保护。这意味着原始的 ARM 指令被转换成了一套自定义的、无法被常规反汇编工具解析的字节码。在这种情况下, 试图完全理解并"白盒"地还原签名算法, 几乎是不可能的。

因此, 业界的分析思路从"算法还原"转向"算法利用", 这便是黑盒 RPC (Remote Procedure Call) 调用。

什么是黑盒 RPC 调用?

核心思想是：不再关心函数内部是如何实现的，而是将其作为一个整体，一个黑盒子。我们只关心它的输入和输出。

我们将通过 Frida 等工具，在 App 的运行时环境中，强行调用这个黑盒函数，让它为我们计算出所需的结果（如 `sign`, `wua`），然后将结果返回给外部的自动化程序。

这就好比我们使用一个网站的 API，我们不需要它的源码，只需要知道它的 URL、参数和返回值格式就能使用它。在这里，SO 里的函数就是那个"API"。

实现黑盒 RPC 的核心步骤

第一步：定位目标函数地址 (Finding the Function Pointer)

这是最困难、最耗时的一步，需要深厚的动态调试功底。

1. 从 JNI 入口开始：从 Java 层调用 Native 方法的地方（JNI 函数）作为起点。
2. 主动调用与插桩：使用 Frida 主动调用该 JNI 函数，并使用 Stalker 等指令级跟踪工具，记录下执行轨迹。
3. 执行流分析：分析 Stalker 产生的巨大日志，或使用 Unicorn Engine 等模拟执行工具进行分析，理清复杂的跳转和计算逻辑，最终找到一个"干净"的函数入口——它接收相对原始的业务参数，返回最终的签名结果。这个函数的地址（通常是 SO 基址 + 偏移量）就是我们的目标。

第二步：分析函数原型 (Analyzing the Prototype)

确定目标函数的输入和输出。

- 输入参数：在上一步找到的函数调用点下断点，观察调用前各寄存器（ARM32 下重点关注 R0-R3）和栈上的值，推断出函数的参数类型、数量和顺序。参数可能是字符串、字节数组、结构体指针等。
- 返回值：在函数返回点下断点，观察 R0 寄存器的值，确定函数的返回值是什么（通常是一个指向结果字符串的指针或一个状态码）。

第三步：构建 RPC 服务端 (Frida Agent)

编写一个 Frida 脚本，将定位到的原生函数封装成一个可供远程调用的服务。

agent.js:

```
// 1. Get SO base address
const baseAddr = Module.findBaseAddress("libsgmain.so");
// 2. Calculate target function absolute address
// This 0x123456 is the function offset found through great effort in step one
const targetFuncPtr = baseAddr.add(0x123456);

// 3. Define function based on the prototype analyzed in step two
// Assume function prototype is: char* func(char* input1, int input2)
const nativeFunc = new NativeFunction(targetFuncPtr, "pointer", [
    "pointer",
    "int",
]);

// 4. Expose interface using rpc.exports
rpc.exports = {
    // Define a remote call interface named getSign
    getSign: function (param1, param2) {
        console.log("RPC call received, invoking native function...");
        // Prepare parameters for native function
        const input1Ptr = Memory.allocUtf8String(param1);

        // Call native function
        const resultPtr = nativeFunc(input1Ptr, param2);

        // Read and return result
        return resultPtr.readUtf8String();
    },
};
```

client.py:

```
import frida

def main():
    # Connect to frida-server on device
    device = frida.get_usb_device()

    # Attach to target App process
    pid = device.spawn(["com.example.app"])
    session = device.attach(pid)

    # Load Frida Agent script
    with open("agent.js") as f:
        script_code = f.read()
    script = session.create_script(script_code)
    script.load()

    # Prepare parameters
    api_params_str = "param1=value1&param2=value2"
    some_int_value = 123

    # Call RPC interface like calling a local function
    print("Calling RPC function: getSign...")
    result_sign = script.exports.get_sign(api_params_str, some_int_value)

    print(f"Successfully got sign: {result_sign}")

    # Can use the obtained sign to construct and send network requests here
    # ...

    session.detach()

if __name__ == '__main__':
    main()
```

通过黑盒 RPC，我们可以绕过对 VMP 等复杂技术的直接对抗，将逆向的重点放在寻找和调用关键函数上，这在当今的高级移动安全攻防中是一种务实且高效的策略。

[C05] 视频 App 与 DRM 案例

C05: 视频 App 与 DRM 案例

视频类 App 的逆向分析是移动端安全领域最具挑战性的方向之一，其核心难点在于数字版权管理（DRM）技术的对抗。本案例将深入探讨视频 App，特别是涉及 DRM 的分析思路。

核心分析目标

1. 视频流分析: 解析视频播放的网络协议, 如 `HLS` (`.m3u8`) 和 `DASH` (`.mpd`), 并提取视频分片。
2. 解锁 VIP 功能: 绕过付费墙, 观看 VIP 专属影片或解锁更高清晰度 (如 1080p, 4K)。
3. DRM 对抗: 理解 DRM 的工作原理, 并尝试获取解密视频所需的密钥。 (注意: 这通常是极其困难的, 且可能涉及法律风险。)

案例：分析一个使用 Widevine DRM 的视频播放流程

第 1 步：视频流协议分析

目标: 找到描述视频信息的清单文件 (`.m3u8` 或 `.mpd`)。

1. 网络抓包: 打开 Charles 或 Mitmproxy, 启动目标视频 App 并播放一个影片。
2. 过滤请求: 在抓包结果中, 使用关键词 `m3u8` 或 `mpd` 进行过滤。你很快就能定位到一个请求, 其 URL 类似于 `https://.../video.mpd`。
3. 分析清单文件:
 - DASH (`.mpd`): 这是一个 XML 文件, 描述了视频的各种信息, 包括不同的分辨率、音轨、字幕轨道以及加密信息。

- HLS (.m3u8): 这是一个文本文件。主 m3u8 文件可能指向多个子 m3u8 文件，每个子文件代表一种特定的码率（清晰度），并包含了该码率下所有视频分片（.ts 文件）的 URL。

在清单文件中，你会找到一个关键的标签，表明内容是受保护的，例如：

```
<!-- DASH MPD inEncryptInformation -->
<ContentProtection schemeIdUri="urn:uuid:edef8ba9-79d6-4ace-a3c8-27dc51d21ed"
cenc:default_KID="...value...">
<cenc:pssh>...</cenc:pssh>
</ContentProtection>
```

第 2 步：理解 DRM 工作流程 (Widevine)

Google 的 Widevine 是 Android 平台上最主流的 DRM 方案。它分为三个安全级别 (L1, L2, L3)，其中 L1 安全性最高。

1. App 请求播放: App 从视频清单中解析出 pssh 数据。
2. 获取许可证 (License): App 将 pssh 数据发送给系统的 MediaDrm API，生成一个许可证请求 (License Request)。然后，App 将这个请求发送到视频服务提供商的许可证服务器。
3. 服务器验证: 许可证服务器验证请求的合法性（例如，验证用户的 VIP 身份），然后返回一个加密的许可证 (Encrypted License)。
4. 解密密钥: App 将加密的许可证提供给 MediaDrm API。这一步是关键：
 - L1 安全级别: 许可证的处理和内容密钥的解密完全在处理器的可信执行环境 (TEE) 中进行。Android 操作系统和 App 本身都无法访问到解密后的密钥。视频帧的解密也在 TEE 中完成，然后直接输出到屏幕，不会在 App 的内存中暴露。
 - L3 安全级别: 在没有 TEE 支持的设备上，这些操作都在软件层面完成。因此，L3 是理论上最容易被攻击的。

第 3 步：逆向分析与信息获取

由于 L1 的硬件级保护，直接获取内容密钥（Content Key）几乎是不可能的，但在一些 arxiv 论文中，笔者确实找到了在一些特定设备中可以利用的漏洞。因此，分析的重点转向了许可证的获取过程。

目标：拦截 App 与许可证服务器之间的通信，获取许可证请求和响应。

1. 定位许可证请求代码：

- 搜索 `MediaDrm`, `getKeyRequest`, `provideKeyResponse` 等 `android.media` 包中的 DRM 相关 API。
- 使用 Frida Hook 这些方法，可以打印出 `pssh`、许可证请求和加密的许可证响应。

```
Java.perform(function () {
    var MediaDrm = Java.use("android.media.MediaDrm");

    // Hook getKeyRequest method to capture license requests
    MediaDrm.getKeyRequest.implementation = function (
        scope,
        initData,
        mimeType,
        keyType,
        optionalParameters
    ) {
        console.log("Intercepting getKeyRequest...");
        // initData is the pssh
        console.log("PSSH (initData):", bytesToHex(initData));

        var keyRequest = this.getKeyRequest(
            scope,
            initData,
            mimeType,
            keyType,
            optionalParameters
        );
        // keyRequest is a complex object, needs further parsing
        // ...
        return keyRequest;
    };

    // Hook provideKeyResponse method to capture encrypted license
    MediaDrm.provideKeyResponse.implementation = function (scope, response) {
        console.log("Intercepting provideKeyResponse...");
        // response is the encrypted license obtained from the server
        console.log("Encrypted License (response):", bytesToHex(response));

        return this.provideKeyResponse(scope, response);
    };
});
function bytesToHex(arr) {
    /* ... a function to convert byte array to hex string ... */
}
```

1. CDM (内容解密模块) 分析: Widevine 的 L3 级 CDM 是一个原生库 (.so 文件)，负责处理白盒加密的逻辑。对这个 .so 文件进行深入的静态和动态分析，是理论上还原出设备密钥 (Device Key) 的唯一途径，这也是 CDM Challenge 等比赛的核心。这是一个极其复杂和耗时的过程。

主流平台 DRM 与加密方案实例

国内平台 (某酷、某奇艺、某讯视频、某果 TV)

国内主流视频平台在加密策略上通常采用"自研加密方案 + 标准 DRM"的混合模式。

- 对于拥有全球版权的影视剧（如好莱坞大片），它们会使用行业标准的 Widevine DRM。
- 对于大量的自制剧、综艺等内容，它们更倾向于使用自研的加密方案，其核心是对 HLS 协议进行改造。

通用模式：保护 HLS 密钥的获取过程

1. 视频流：普遍使用 HLS (.m3u8) 协议。
2. 加密算法：.m3u8 文件中会声明视频分片 (.ts 文件) 使用 AES-128-CBC 加密。
3. 核心保护：视频数据本身的加密算法是标准的，但获取解密密钥（Key）的过程是高度定制和保护的。
 - .m3u8 文件本身不是静态的，而是通过一个需要复杂签名的 API 动态生成的。
 - #EXT-X-KEY 标签中指向的密钥 URL (key.key) 也不是一个能直接访问的地址，访问它同样需要正确的 Cookie、Referer 和加密参数。

1. 逆向关键：

- 定位播放 API：逆向的重点是找到 App 中负责请求视频播放信息的 API。这个 API 的请求参数通常包含视频 ID、清晰度、以及一个类似我们在上一章分析过的、包含设备指纹和时间戳的 sign 或 token。
- 模拟合法请求：只要能够成功模拟这个 API 的调用，就能获取到一个包含了有效密钥 URL 的 .m3u8 文件。拿到密钥后，就可以使用标准的 AES-128 算法解密 .ts 文件并合并成一个完整的视频。
- 某讯视频的 vkey：一个典型的例子是某讯视频，其播放 API 中需要一个至关重要的 vkey 参数，这个参数的生成算法就封装在客户端的 SO 库中。

国外平台 (Netflix, 某管, Hulu, HBO Max)

国外主流视频平台，特别是内容提供商，严格且深度地依赖标准化的 DRM 体系。逆向的焦点完全不在于分析视频文件格式或算法，而在于 DRM 许可证的获取流程。

Netflix / Hulu / HBO Max

- DRM 方案：在 Android 上无一例外地使用 Google Widevine，在苹果设备上使用 FairPlay。
- 安全级别：对于高清内容 (HD, 4K)，强制要求设备的 Widevine 安全级别为 L1。这意味着密钥交换和内容解密全程在硬件 TEE 中完成，App 和操作系统均无法触及明文密钥。
- 许可证请求保护：逆向的唯一着眼点是 App 发起许可证请求的过程。
- 这个请求被多种方式保护，例如 Netflix 使用自研的 MSL (Message Security Layer) 协议对许可证请求本身进行二次封装和加密。
- App 会采集大量设备指纹信息，连同用户的身份凭证一起，用于生成许可证请求。服务端的风控系统会严格校验这些信息，以确保请求来自于一个合法的、未被篡改的官方 App 客户端。
- 逆向结论：在 L1 保护下，通过逆向 App 来获取视频解密密钥以进行离线下载是几乎不可能的。分析的主要意义在于理解其架构和安全强度。

某管

某管的情况比较特殊，它需要区分对待：

- 付费内容 (某管 Premium / 电影)：与 Netflix 类似，使用标准的 Widevine DRM 进行保护。
- 普通 UGC 内容：大部分视频没有使用 DRM 加密，但使用了另一种巧妙的保护方式——动态 URL 签名。
- 现象：使用 `yt-dlp` 等工具下载视频时，会看到它有一个"deciphering signature"的过程。

- 原理: 视频流的 URL 中包含一个 `s` 或 `sig` 参数, 这个签名是由一段混淆过的 JavaScript 代码 (在 Web 端) 或 Native 代码 (在 App 端) 动态生成的。该算法将视频的 `cipher` (一段加密字符串) 和其他参数作为输入, 输出一个解密的签名。
- 逆向关键: 逆向的重点不再是 DRM, 而是找到并还原那段负责计算签名的 JavaScript/Native 函数。由于代码经过了高度混淆, 这依然是一项具有挑战性的工作。

音乐流媒体平台 (某果音乐, 某破天)

音乐流媒体平台同样采用 DRM 保护其内容, 但由于音频文件体积较小、交互模式不同, 其加密方案与视频平台有所差异。

某果音乐

某果音乐采用 Widevine DRM + HLS 的组合方案保护其音频内容。

认证体系:

- `accessToken`: 从 Web 端 JavaScript 中提取的 JWT Token, 用于 API 认证
- `mediaUserToken`: 用户身份令牌, 存储在 Cookie 中, 用于访问个人化内容和高品质音源

API 架构:

```
class AppleMusicAPI:  
    # 元数据 API - 获取专辑、歌曲、播放列表信息  
    METADATA_API = "https://amp-api.music.apple.com/v1/catalog/{storefront}/{type}s/  
    {id}"  
  
    # 播放信息 API - 获取音频流地址和加密参数  
    WEBPLAYBACK_API = "https://play.itunes.apple.com/WebObjects/MZPlay.woa/wa/  
    webPlayback"  
  
    def get_webplayback(self, adam_id):  
        """获取音频播放信息"""  
        return self.session.post(  
            self.WEBPLAYBACK_API,  
            json={'salableAdamId': adam_id}  
        ).json()["songList"][0]
```

DRM 解密流程:

1. 从 webPlayback 接口获取 HLS 播放列表 URL
2. 解析 `.m3u8` 文件, 提取 `#EXT-X-KEY` 中的 PSSH 数据
3. 使用 Widevine CDM 构造 License Challenge
4. 向 `hls-key-server-url` 发送许可证请求
5. 解析 License 响应, 提取 AES 解密密钥
6. 使用密钥解密音频分片

```
def get_decryption_key(self, song_id, pssh):  
    """获取音频解密密钥"""  
    # 构造Widevine PSSH 对象  
    widevine_pssh = WidevinePsshData()  
    widevine_pssh.algorithm = 1  
    widevine_pssh.key_ids.append(base64.b64decode(pssh.split(",")[1]))  
  
    # 获取 License Challenge  
    pssh_obj = PSSH(widevine_pssh.SerializeToString())  
    session = self.cdm.open()  
    challenge = base64.b64encode(  
        self.cdm.get_license_challenge(session, pssh_obj)  
    ).decode()  
  
    # 请求 License  
    license_resp = self.session.post(  
        self.license_url,  
        json={  
            "admid": song_id,  
            "challenge": challenge,  
            "key-system": "com.widevine.alpha",  
            "uri": f"data:;base64,{pssh}",  
        }  
    ).json()  
  
    # 解析密钥  
    self.cdm.parse_license(session, license_resp["license"])  
    return next(  
        k for k in self.cdm.get_keys(session)  
        if k.type == "CONTENT"  
    ).key.hex()
```

音频格式: 主要使用 `28:ctrp256` 格式 (256kbps AAC), 通过 HLS 协议分发。

某破天

某破天 采用双轨加密方案，根据音频格式使用不同的加密机制。

认证体系：

- sp_dc Cookie: 用户会话凭证，从浏览器 Cookie 中提取
- TOTP 令牌: 时间动态令牌，用于生成 accessToken (详见 [TOTP 技术原理](#))
- Client Token: 设备认证令牌，包含设备指纹信息

TOTP 认证机制 (关键反爬措施)：

```
class MusicTOTP:  
    """某破天 使用动态 TOTP 进行 API 认证"""  
  
    def __init__(self):  
        # 从远程获取 TOTP 参数 (需要定期更新)  
        self.version = self._fetch_param("VERSION")  
        self.period = self._fetch_param("PERIOD")  
        self.digits = self._fetch_param("DIGITS")  
        self.secret = self._derive_secret()  
  
    def generate(self, timestamp_ms):  
        """生成 TOTP 令牌"""  
        counter = timestamp_ms // 1000 // self.period  
        counter_bytes = counter.to_bytes(8, byteorder="big")  
  
        h = hmac.new(self.secret, counter_bytes, hashlib.sha1)  
        hmac_result = h.digest()  
  
        offset = hmac_result[-1] & 0x0F  
        binary = (  
            (hmac_result[offset] & 0x7F) << 24 |  
            (hmac_result[offset + 1] & 0xFF) << 16 |  
            (hmac_result[offset + 2] & 0xFF) << 8 |  
            (hmac_result[offset + 3] & 0xFF)  
        )  
        return str(binary % (10 ** self.digits)).zfill(self.digits)
```

API 架构：

```
class MusicAPI:  
    # 元数据 API  
    METADATA_API = "https://api.example.com/v1/{type}/{item_id}"  
  
    # 播放信息 API  
    PLAYBACK_API = "https://gue1-spclient.example.com/track-playback/v1/media/{type}:  
{id}"  
  
    # 流地址解析 API  
    STREAM_API = "https://gue1-spclient.example.com/storage-resolve/v2/files/audio/  
interactive/11/{file_id}"  
  
    # Widevine 许可证 API (AAC 格式)  
    WIDEVINE_LICENSE_API = "https://gue1-spclient.example.com/widevine-license/v1/  
{type}/license"  
  
    # PlayPlay 许可证 API (Vorbis 格式)  
    PLAYPLAY_LICENSE_API = "https://gew4-spclient.example.com/playplay/v1/key/  
{file_id}"
```

双轨加密方案:

格式	加密方式	品质	密钥获取
OGG Vorbis	PlayPlay (AES-CTR)	96/160/320 kbps	PlayPlay API
AAC (M4A)	Widevine DRM	128/256 kbps	Widevine License API

PlayPlay 解密 (Vorbis 格式):

```
def decrypt_playplay(self, decryption_key, encrypted_path, decrypted_path):
    """某破天自研PlayPlay加密使用AES-CTR模式"""
    cipher = AES.new(
        decryption_key,
        AES.MODE_CTR,
        nonce=bytes.fromhex("72e067fbddcb****"),
        # 固定Nonce
        initial_value=bytes.fromhex("ebe8bc643f63****"),
        # 固定IV
    )

    with open(encrypted_path, "rb") as f:
        encrypted_data = f.read()

    decrypted_data = cipher.decrypt(encrypted_data)

    # 查找OGG文件头
    offset = decrypted_data.find(b"OggS")
    with open(decrypted_path, "wb") as f:
        f.write(decrypted_data[offset:])
```

JA3 指纹检测绕过:

某破天 使用 JA3 指纹检测来识别非官方客户端。绕过方案:

```
from curl_cffi import requests as curl_requests

# 使用curl_cffi模拟Chrome的TLS指纹
session = curl_requests.Session(impersonate="chrome120")
```

无头浏览器保活 (Token 自动刷新):

某破天 的 accessToken 有效期较短, 且 TOTP 认证可能被风控拦截。使用无头浏览器可以模拟真实用户行为, 自动获取新的 Token:

```
class HeadlessTokenFetcher:  
    """使用无头浏览器自动获取 某破天 Token"""  
  
    def __init__(self):  
        self.token_file = Path.home() / '.music_token.json'  
        self.cookies_file = Path('./cookies.txt')  
  
    def fetch_with_playwright(self):  
        """使用 Playwright 无头浏览器获取 Token"""  
        from playwright.sync_api import sync_playwright  
  
        with sync_playwright() as p:  
            browser = p.chromium.launch(  
                headless=True,  
                args=[  
                    '--no-sandbox',  
                    '--disable-blink-features=AutomationControlled'  
                ]  
            )  
  
            context = browser.new_context(  
                user_agent='Mozilla/5.0 (Windows NT 10.0; Win64; x64) '  
                'AppleWebKit/537.36 Chrome/142.0.0.0 Safari/537.36'  
            )  
  
            # 加载预存的 Cookies (包含 sp_dc)  
            if self.cookies_file.exists():  
                cookies = self._load_cookies()  
                context.add_cookies(cookies)  
  
            page = context.new_page()  
            token_data = {}  
  
            # 拦截网络响应，捕获 Token  
            def handle_response(response):  
                nonlocal token_data  
                if '/api/token' in response.url and 'reason=init' in response.url:  
                    data = response.json()  
                    if 'accessToken' in data:  
                        token_data = data  
  
            page.on('response', handle_response)  
  
            # 访问某破天 触发 Token 请求  
            page.goto('https://open.example.com', wait_until='networkidle')  
            page.wait_for_timeout(3000)  
  
            browser.close()  
  
            if token_data:  
                self._save_token(token_data)
```

```
return True  
return False
```

保活策略:

1. 定时任务每 30-50 分钟刷新 Token (有效期约 1 小时)
2. 在 License 请求失败 (403) 时自动触发刷新
3. 使用 Cookies 而非账号密码, 避免登录风控
4. 模拟真实浏览器指纹, 绕过自动化检测

逆向要点:

1. TOTP 参数需要动态获取, 某破天 会定期更新
2. 高品质音源 (320kbps Vorbis, 256kbps AAC) 需要 Premium 账户
3. PlayPlay 的 Nonce 和 IV 是固定的, 但密钥是动态获取的
4. Client Token 包含设备指纹, 用于风控检测
5. 建议使用无头浏览器保活方案, 比纯 API 方式更稳定

视频平台 API 签名实战

以下是基于 yt-dlp 项目的真实实现分析, 展示了如何逆向还原各平台的视频获取逻辑。

B 某站 WBI 签名机制

B 某站使用 WBI (Web 接口签名) 机制保护其 API 接口。

WBI 密钥获取与签名

```
class BiliBaseIE:
    _WBI_KEY_CACHE_TIMEOUT = 30 # 密钥缓存 30 秒

    def _get_wbi_key(self, video_id):
        """从 nav 接口获取 WBI 签名密钥"""
        if time.time() < self._wbi_key_cache.get('ts', 0) + self._WBI_KEY_CACHE_TIMEOUT:
            return self._wbi_key_cache['key']

        # 获取 img_url 和 sub_url 中的文件名
        session_data = self._download_json(
            'https://api.example.com/x/web-interface/nav', video_id,
            note='Downloading wbi sign')

        lookup = ''.join(traverse_obj(session_data, (
            'data', 'wbi_img', ('img_url', 'sub_url'),
            {lambda x: x.rpartition('/')[2].partition('.')[0]})))

        # getMixinKey() 的置换表 - 从 B某站 前端 JS 逆向得到
        mixin_key_enc_tab = [
            46, 47, 18, 2, 53, 8, 23, 32, 15, 50, 10, 31, 58, 3, 45, 35,
            27, 43, 5, 49, 33, 9, 42, 19, 29, 28, 14, 39, 12, 38, 41, 13,
            37, 48, 7, 16, 24, 55, 40, 61, 26, 17, 0, 1, 60, 51, 30, 4,
            22, 25, 54, 21, 56, 59, 6, 63, 57, 62, 11, 36, 20, 34, 44, 52,
        ]

        # 通过置换表生成最终密钥
        self._wbi_key_cache.update({
            'key': ''.join(lookup[i] for i in mixin_key_enc_tab)[:32],
            'ts': time.time(),
        })
        return self._wbi_key_cache['key']

    def _sign_wbi(self, params, video_id):
        """对请求参数进行 WBI 签名"""
        params['wts'] = round(time.time())
        # 过滤特殊字符
        params = {
            k: ''.join(filter(lambda char: char not in "!'()*", str(v)))
            for k, v in sorted(params.items())
        }
        query = urllib.parse.urlencode(params)
        # 计算 w_rid = md5(query + wbi_key)
        params['w_rid'] = hashlib.md5(
            f'{query}{self._get_wbi_key(video_id)}'.encode()
        ).hexdigest()
        return params
```

视频格式提取

```
def extract_formats(self, play_info):
    """从 playurl 接口提取视频格式"""
    formats = []

    # 提取音频轨道 (DASH)
    audios = traverse_obj(play_info, ('dash', (None, 'dolby'), 'audio', ...))
    flac_audio = traverse_obj(play_info, ('dash', 'flac', 'audio'))
    if flac_audio:
        audios.append(flac_audio)

    for audio in audios:
        formats.append({
            'url': traverse_obj(audio, 'baseUrl', 'base_url', 'url'),
            'acodec': traverse_obj(audio, ('codecs', {str.lower})),
            'vcodec': 'none',
            'tbr': float_or_none(audio.get('bandwidth'), scale=1000),
        })

    # 提取视频轨道 (DASH)
    for video in traverse_obj(play_info, ('dash', 'video', ...)):
        formats.append({
            'url': traverse_obj(video, 'baseUrl', 'base_url', 'url'),
            'fps': float_or_none(traverse_obj(video, 'frameRate', 'frame_rate')),
            'width': int_or_none(video.get('width')),
            'height': int_or_none(video.get('height')),
            'vcodec': video.get('codecs'),
            'acodec': 'none',
            'dynamic_range': {126: 'DV', 125: 'HDR10'}.get(video.get('id')),
        })

    return formats
```

某奇艺 SDK 签名算法

某奇艺使用复杂的 SDK 进行请求签名，其核心是多轮 MD5 变换。

SDK 解释器实现

```
class VideoSDK:  
    def __init__(self, target, ip, timestamp):  
        self.target = target  
        self.ip = ip  
        self.timestamp = timestamp  
  
    @staticmethod  
    def split_sum(data):  
        """将十六进制字符串每个字符转为整数求和"""  
        return str(sum(int(p, 16) for p in data))  
  
    @staticmethod  
    def digit_sum(num):  
        """计算数字各位之和"""  
        return str(sum(map(int, str(num))))  
  
    def even_odd(self):  
        """分离时间戳的奇偶位"""  
        even = self.digit_sum(str(self.timestamp)[:2])  
        odd = self.digit_sum(str(self.timestamp)[1:2])  
        return even, odd  
  
    def mod(self, modulus):  
        """对 IP 地址取模并拼接"""  
        self.target = hashlib.md5(self.target.encode()).hexdigest()  
        chunks = [self.target[i*32:(i+1)*32] for i in range(1)]  
        ip = list(map(int, self.ip.split('.')))  
        self.target = chunks[0] + ''.join(str(p % modulus) for p in ip)  
  
    def split(self, chunksize):  
        """按块大小分割并与 IP 混合"""  
        modulus_map = {4: 256, 5: 10, 8: 100}  
        self.target = hashlib.md5(self.target.encode()).hexdigest()  
        chunks = [self.target[i*chunksize:(i+1)*chunksize]  
                  for i in range(32 // chunksize)]  
        ip = list(map(int, self.ip.split('.')))  
        ret = ''  
        for i, chunk in enumerate(chunks):  
            ip_part = str(ip[i] % modulus_map[chunksize]) if i < 4 else ''  
            ret += (ip_part + chunk) if chunksize == 8 else (chunk + ip_part)  
        self.target = ret  
  
    def handle_input16(self):  
        """16 字节分割处理"""  
        self.target = hashlib.md5(self.target.encode()).hexdigest()  
        self.target = (self.split_sum(self.target[:16]) +  
                      self.target +  
                      self.split_sum(self.target[16:])))  
  
    def date(self, scheme):  
        """日期混合 (y/m/d 组合)"""  
        self.target = hashlib.md5(self.target.encode()).hexdigest()
```

```

d = time.localtime(self.timestamp)
strings = {'y': str(d.tm_year), 'm': f'{d.tm_mon:02d}', 'd':
f'{d.tm_mday:02d}'}
self.target += ''.join(strings[c] for c in scheme)

class VideoSDKInterpreter:
    """动态解释执行从服务器获取的 SDK 代码"""
    def __init__(self, sdk_code):
        self.sdk_code = sdk_code

    def run(self, target, ip, timestamp):
        # 解码混淆的 JS 代码
        self.sdk_code = decode_packed_codes(self.sdk_code)
        # 提取函数调用序列
        functions = re.findall(r'input=([a-zA-Z0-9]+)\(input', self.sdk_code)

        sdk = VideoSDK(target, ip, timestamp)

        for function in functions:
            if re.match(r'mod\d+', function):
                sdk.mod(int(function[3:]))
            elif re.match(r'date[ymd]{3}', function):
                sdk.date(function[4:])
            elif re.match(r'split\d+', function):
                sdk.split(int(function[5:]))
            # ... 其他函数映射

        return sdk.target

```

登录与密钥获取

```

def get_raw_data(self, tvid, video_id):
    """获取视频播放数据"""
    tm = int(time.time() * 1000)
    key = 'd5fb4bd9d50c4be6948c97edd7254***' # 硬编码密钥
    sc = hashlib.md5((str(tm) + key + tvid).encode()).hexdigest()

    params = {
        'tvid': tvid,
        'vid': video_id,
        'src': '76f90cbd92f94a2e925d83e8ccd22***',
        'sc': sc,
        't': tm,
    }
    return self._download_json(api_url, video_id, query=params)

```

某讯视频 cKey 加密

某讯视频使用 AES-CBC 加密生成 cKey 参数。

cKey 生成算法

```
from aes import aes_cbc_encrypt_bytes

class TencentBaseIE:
    _APP_VERSION = '3.5.57'
    _PLATFORM = '10901'

    def _get_ckey(self, video_id, url, guid):
        """生成 cKey 加密参数"""
        ua = self.get_param('http_headers')['User-Agent']

        # 构造 payload: vid|timestamp|magic|version|guid|platform|url|ua||Mozilla|...
        payload = (
            f'{video_id}|{int(time.time())}|mg3c3b04ba|{self._APP_VERSION}|'
            f'{guid}|{self._PLATFORM}|{url[:48]}|{ua.lower()[:48]}||Mozilla|'
            f'Netscape|Windows x86_64|00|'
        )

        # AES-CBC 加密
        # Key: 0k\xda\xa3\x9e/\x8c\xb0\x7f^r-\x9e\xde\x**\x**
        # IV: \x01PJ\xf3V\xe6\x19\xcf.B\xbb\xa6\x**?**\x**
        encrypted = aes_cbc_encrypt_bytes(
            bytes(f'{sum(map(ord, payload))}|{payload}', 'utf-8'),
            b'0k\xda\xa3\x9e/\x8c\xb0\x7f^r-\x9e\xde\x**\x**',
            b'\x01PJ\xf3V\xe6\x19\xcf.B\xbb\xa6\x**?**\x**',
            padding_mode='whitespace'
        )
        return encrypted.hex().upper()

    def _get_video_api_response(self, video_url, video_id, series_id, ...):
        """请求视频 API"""
        guid = ''.join(random.choices(string.digits + string.ascii_lowercase, k=16))
        ckey = self._get_ckey(video_id, video_url, guid)

        query = {
            'vid': video_id,
            'cid': series_id,
            'cKey': ckey,
            'encryptVer': '8.1',
            'sphls': '2',          # HLS 格式
            'dtype': '3',
            'defn': video_quality,
            'sphttps': '1',        # 启用 HTTPS
            'hevclv': '28',        # 启用 HEVC
            'drm': '40',           # DRM 标识
            'guid': guid,
            'flowid': ''.join(random.choices(string.ascii_lowercase + string.digits,
                k=32)),
        }
        return self._download_json(self._API_URL, video_id, query=query)
```

某酷 CNA 设备标识

某酷使用 CNA (Client Network Address) 作为设备标识。

```
class VideoIE:  
    @staticmethod  
    def get_ysuid():  
        """生成会话 ID"""  
        return f'{int(time.time())}{''.join(random.choices(string.ascii_letters,  
k=3))}'  
  
    def _real_extract(self, url):  
        video_id = self._match_id(url)  
  
        # 设置必要的 Cookie  
        self._set_cookie('example.com', '__ysuid', self.get_ysuid())  
        self._set_cookie('example.com', 'xreferrer', 'http://www.example.com')  
  
        # 从mmstat.com 获取CNA (通过ETag头)  
        _, urlh = self._download_webpage_handle(  
            'https://log.mmstat.com/eg.js', video_id, 'Retrieving cna info')  
        cna = urlh.headers['etag'][1:-1] # 去除双引号  
  
        # 构造API请求参数  
        params = {  
            'vid': video_id,  
            'ccode': '0564',  
            'client_ip': '192.168.1.1',  
            'utid': cna, # CNA 作为utid  
            'client_ts': time.time() / 1000,  
        }  
  
        data = self._download_json(  
            'https://ups.example.com/ups/get.json', video_id,  
            query=params, headers={'Referer': url})['data']  
  
        # 提取m3u8格式  
        formats = [  
            {'url': stream['m3u8_url'],  
             'format_id': self.get_format_name(stream.get('stream_type')),  
             'ext': 'mp4',  
             'protocol': 'm3u8_native',  
             'width': stream.get('width'),  
             'height': stream.get('height'),  
            } for stream in data['stream']]  
  
        return {'id': video_id, 'formats': formats, ...}
```

某果 TV tk2 令牌

某果 TV 使用 Base64 编码的设备信息 作为令牌。

```
class MGTVIE:
    def _real_extract(self, url):
        video_id = self._match_id(url)

        # 生成 tk2 令牌: Base64(did=uuid|pno=1030|ver=0.3.0301|clit=timestamp)
        tk2 = base64.urlsafe_b64encode(
            f'did={uuid.uuid4()}|pno=1030|ver=0.3.0301|
            clit={int(time.time())}'.encode()
        )[::-1] # 反转字节序

        # 第一步: 获取 pm2 令牌
        api_data = self._download_json(
            'https://pcweb.api.example.com/player/video', video_id,
            query={'tk2': tk2, 'video_id': video_id, 'type': 'pch5'})['data']

        # 第二步: 使用 pm2 获取流地址
        stream_data = self._download_json(
            'https://pcweb.api.example.com/player/getSource', video_id,
            query={
                'tk2': tk2,
                'pm2': api_data['atc']['pm2'], # 第一步获取的令牌
                'video_id': video_id,
                'type': 'pch5',
                'src': 'intelmgtv',
            })['data']

        # 提取 HLS 流
        stream_domain = traverse_obj(stream_data, ('stream_domain', ...),
                                      get_all=False)
        formats = []
        for stream in traverse_obj(stream_data, ('stream', lambda _, v: v['url'])):
            format_info = self._download_json(
                urljoin(stream_domain, stream['url']), video_id)
            formats.append({
                'url': format_info['info'],
                'ext': 'mp4',
                'protocol': 'm3u8_native',
            })
        return {'id': video_id, 'formats': formats}
```

某管 Innertube 客户端体系

某管使用 Innertube API 与多种客户端类型进行通信。

客户端配置

```
INNERTUBE_CLIENTS = {
    'web': {
        'INNERTUBE_CONTEXT': {
            'client': {
                'clientName': 'WEB',
                'clientVersion': '2.20250312.04.00',
            },
        },
        'INNERTUBE_CONTEXT_CLIENT_NAME': 1,
        'PO_TOKEN_REQUIRED_CONTEXTS': ['gvs'], # 需要 PoToken
        'SUPPORTS_COOKIES': True,
    },
    'android': {
        'INNERTUBE_CONTEXT': {
            'client': {
                'clientName': 'ANDROID',
                'clientVersion': '20.10.38',
                'androidSdkVersion': 30,
                'userAgent': 'com.google.android.youtube/20.10.38 (Linux; U; Android
11) gzip',
                'osName': 'Android',
                'osVersion': '11',
            },
        },
        'INNERTUBE_CONTEXT_CLIENT_NAME': 3,
        'REQUIRE_JS_PLAYER': False,
    },
    'ios': {
        'INNERTUBE_CONTEXT': {
            'client': {
                'clientName': 'IOS',
                'clientVersion': '20.10.4',
                'deviceMake': 'Apple',
                'deviceModel': 'iPhone16,2',
                'userAgent': 'com.google.ios.youtube/20.10.4 (iPhone16,2; U; CPU iOS
18_3_2 like Mac OS X;',
            },
        },
        'INNERTUBE_CONTEXT_CLIENT_NAME': 5,
    },
    'tv': {
        'INNERTUBE_CONTEXT': {
            'client': {
                'clientName': 'TVHTML5',
                'clientVersion': '7.20250312.16.00',
                'userAgent': 'Mozilla/5.0 (ChromiumStylePlatform) Cobalt/Version',
            },
        },
        'INNERTUBE_CONTEXT_CLIENT_NAME': 7,
    },
}
```

签名解密机制

某管对视频 URL 使用 动态 JavaScript 签名 保护:

```
from jsinterp import JSInterpreter

class VideoTubeIE:
    _PLAYER_INFO_RE = (
        r'/s/player/(?P<id>[a-zA-Z0-9_-]{8,})/(?:tv-)?player',
        r'/(?P<id>[a-zA-Z0-9_-]{8,})/player.*?/base\.js$',
    )

    def _decrypt_signature(self, s, video_id, player_url):
        """
        解密签名的核心流程:
        1. 下载并缓存 player.js
        2. 使用 JSInterpreter 解析混淆的 JS
        3. 找到签名解密函数并执行
        """
        # 获取 player.js 代码
        player_code = self._download_webpage(player_url, video_id)

        # 使用 JS 解释器解析并执行解密函数
        jsi = JSInterpreter(player_code)

        # 在 player.js 中搜索解密函数入口
        # 函数名是动态混淆的, 需要通过特征匹配
        func_name = self._search_regex(
            r'\.get\("n"\)\)\&&\(b=([a-zA-Z0-9$]+)(?:\[\\d+\])?\(([a-zA-Z0-9]\)\)', 
            player_code, 'Initial JS player n function name')

        decrypted = jsi.call_function(func_name, s)
        return decrypted
```

视频平台逆向要点对比

平台	签名机制	加密方式	设备标识	难度
B 某站	WBI (md5 + 置换表)	无	SESSDATA Cookie	中
某奇艺	SDK (多轮 md5)	RSA 登录	IP + 时间戳	高
某讯视 频	cKey (AES-CBC)	DRM 可选	GUID	高
某酷	CNA (ETag)	无	utid/ysuid	低
某果 TV	tk2 (Base64)	无	UUID	低
某管	Innertube + JS 签名	DRM 可选	PoToken	高
某果音 乐	JWT + mediaUserToken	Widevine (HLS)	设备证书	高
某破天	TOTP + ClientToken	Widevine/ PlayPlay	sp_dc + JA3	高

通用逆向策略

1. 网络层分析

```
# 抓包分析要点
1. 识别视频清单接口 (*.m3u8, *.mpd, getSource, playurl 等)
2. 分析请求头中的签名参数 (sign, ckey, token 等)
3. 追踪 Cookie 和 Session 机制
4. 识别设备指纹参数 (guid, utid, deviceId 等)
```

2. 前端 JS 逆向

```
# 常见混淆手段及应对
1. 变量名混淆 → 根据上下文推断功能
2. 字符串加密 → 找到解密函数，动态执行
3. 控制流平坦化 → 使用 AST 还原
4. 代码压缩 → 使用 beautifier 格式化
```

3. 设备模拟

```
# 模拟客户端请求
headers = {
    'User-Agent': 'com.example.app/1.0.0 (Linux; Android 11)',
    'X-Client-Info': json.dumps({
        'platform': 'android',
        'version': '1.0.0',
        'device_id': generate_device_id(),
    })
}
```

总结

视频 App 的 DRM 逆向是一场与硬件和复杂密码学协议的艰苦斗争。与音乐 App 不同，其核心目标通常不是开发一个"下载器"，而是理解其安全体系的强度和弱点。

- 对于普通分析，重点是拦截和理解信令（清单文件、许可证请求/响应）。
- 对于高级研究，核心是攻击 L3 的 CDM 实现，但这需要极高的逆向工程和密码学知识。

这个领域的攻防水平代表了整个行业安全对抗的顶峰。

通过分析 yt-dlp 项目的实现，我们可以看到：

1. 签名算法多样性：从简单的 MD5 到复杂的多轮变换，各平台都有独特的保护机制
2. 客户端模拟的重要性：正确的 User-Agent 和设备标识是成功请求的关键
3. 动态代码执行：许多平台使用动态生成的混淆代码，需要 JS 解释器来处理

4. Token 时效性: 大多数签名包含时间戳, 需要实时计算而非缓存

[C06] Unity 游戏 (Il2Cpp) 案例

C06: Unity 游戏逆向 (Il2Cpp) 案例

Unity 是目前最流行的移动游戏引擎之一。现代 Unity 游戏通常使用 Il2Cpp 脚本后端，将 C# 代码转换为 C++ 代码并编译为 Native 库 (`libil2cpp.so`)。这使得传统的 Java/Smali 逆向方法失效，需要全新的工具和思路。

核心 architecture 与文件结构

一个典型的 Unity Il2Cpp 游戏包含以下关键文件：

1. `lib/armeabi-v7a/libil2cpp.so`：这是游戏的核心逻辑库。所有的 C# 脚本（玩家控制、游戏逻辑、网络通信）都被编译到了这里。
2. `assets/bin/Data/Managed/global-metadata.dat`：这是 Il2Cpp 的元数据文件。它包含了被转换前的 C# 类名、方法名、字段名以及它们在 `libil2cpp.so` 中的偏移地址。这是逆向的关键钥匙。
3. `lib/armeabi-v7a/libmain.so` (或 `libunity.so`)：Unity 引擎的运行时库，通常不需要修改。

逆向流程

第 1 步：元数据提取 (Metadata Dumping)

由于 `libil2cpp.so` 是剥离了符号表 (stripped) 的二进制文件，直接用 IDA 打开只能看到成千上万个无名函数 (`sub_xxxx`)。我们需要结合 `global-metadata.dat` 来还原这些函数的真实名称。

- 工具: [Il2CppDumper](#)

1. 将 APK 解压，提取出 `libil2cpp.so` 和 `global-metadata.dat`。

2. 运行 `Il2CppDumper.exe <libil2cpp.so> <global-metadata.dat>`。

3. 工具会生成：

- `dump.cs`：还原后的 C# 伪代码，展示了所有类、字段和方法结构。
- `script.py`：用于 IDA Pro 的 Python 脚本，可以自动重命名 IDA 中的函数。
- `ghidra.py`：用于 Ghidra 的脚本。
- `DummyDll/`：生成的空 DLL 文件，可以用 dnSpy 打开查看类结构。

第 2 步：静态分析与定位

使用 `dnSpy` 打开生成的 Dummy DLL，或是直接阅读 `dump.cs`，我们可以像阅读源码一样浏览游戏的类结构。

- 寻找切入点：
- 货币修改：搜索 `Coin`, `Gem`, `Money`, `Currency` 等关键词。寻找 `AddCoin()`, `GetMoney()`, `UpdateCurrency()` 等方法。
- 无敌/高伤害：搜索 `PlayerController`, `BattleManager`, `Health`, `Damage`。寻找 `TakeDamage()`, `OnHit()` 等方法。
- 内购破解：搜索 `IAP`, `Purchase`, `Store`, `Payment`。寻找 `OnPurchaseSuccess()`, `VerifyReceipt()` 等方法。
- 示例：在 `dump.cs` 中找到如下类：

```
public class PlayerData {
    public int coin;
    public int gem;
    public void AddCoin(int amount); // Address: 0x123456
    public void SubCoin(int amount); // Address: 0x123460
}
```

// Il2Cpp Hook Template

```
var soName = "libil2cpp.so"; var baseAddr = Module.findBaseAddress(soName);
```

```
if (baseAddr) { // Target function offset: 0x123456 (AddCoin) var addCoinFunc =  
baseAddr.add(0x123456);  
  
Interceptor.attach(addCoinFunc, { onEnter: function(args) { // args[0] is 'this' pointer  
(PlayerData instance) // args[1] is amount (coin count to add)  
  
console.log(" [*] AddCoin called"); console.log(" Amount: " + args[1].toInt32());  
  
// Modify parameter: force add 99999 regardless of game logic args[1] =  
ptr(99999); }, onLeave: function(retval) { console.log(" [*] AddCoin finished"); } }); } else  
{ console.log(" [-] libil2cpp.so not found!"); }
```

```
// Use frida-il2cpp-bridge  
Il2Cpp.perform(() => {  
    // 1. Find class  
    const PlayerData = Il2Cpp.domain.assembly("Assembly-  
CSharp").image.class("PlayerData");  
  
    // 2. Hook method (auto process offset, no need to calculate manually)  
    PlayerData.method("SubCoin").implementation = function (amount) {  
        console.log(" [*] SubCoin called with amount: " + amount);  
        // Prevent coin deduction (do nothing)  
        return;  
    };  
  
    // 3. Manually call method  
    // Assume we want to call PlayerData.Instance.AddCoin(1000)  
    // Need to find static instance or current instance first  
  
    // Trace all PlayerData instance creation  
    Il2Cpp.traceClass(PlayerData);  
});
```

- 对抗:

- Hook 加载函数: 游戏必须在运行时解密 metadata 才能正常运行。Hook `libil2cpp.so` 中加载 metadata 的函数（通常是 `il2cpp::vm::MetadataCache::Register` 或相关初始化函数），Dump 出解密后的内存内容。
- 分析解密逻辑: 逆向 `libil2cpp.so` 的初始化流程，找到解密 metadata 的算法（通常是 XOR 或 AES），写脚本还原。

2. 函数地址混淆 / 动态计算

- 现象: ll2CppDumper 导出的地址与内存中的实际地址不符。
- 对抗:
 - 这通常是因为游戏在运行时动态修改了函数指针。
 - 使用 Frida 的扫描功能, 根据机器码特征 (Pattern Scanning) 来定位函数, 而不是依赖固定的偏移。

3. 反调试与完整性校验

- 现象: 附加 Frida 后游戏崩溃或闪退。
- 对抗:
 - 参考 "Anti-Debugging" 章节, 隐藏 Frida 特征, Bypass TracerPid 检测。
 - 使用 Magisk + Riru + ll2CppDumper (Zygisk 版) 在系统层面进行 Dump, 规避应用层检测。

总结

Unity ll2Cpp 逆向的核心在于还原符号。只要拿到了正确的 `global-metadata.dat` 和 `libil2cpp.so` 的映射关系, 剩下的工作就变成了标准的逻辑分析和 Native Hook。熟练掌握 ll2CppDumper 和 Frida 是搞定这类游戏的关键。

[C07] Flutter 应用案例

C07: Flutter 应用逆向案例

Flutter 是 Google 推出的跨平台 UI 框架，它使用 Dart 语言开发。与传统的 Android App (Java/Kotlin) 或 Unity (C#) 不同，Release 模式下的 Flutter 应用将 Dart 代码预编译 (AOT) 成了原生机器码，打包在 `libapp.so` 中，这使得逆向难度大大增加。

核心架构

1. `lib/armeabi-v7a/libflutter.so` : Flutter 引擎，负责渲染、通信和运行时支持。通常不需要逆向，但可以用它来定位关键的内部函数。
2. `lib/armeabi-v7a/libapp.so` : 逆向的核心目标。包含了开发者的所有业务逻辑代码 (Dart 代码编译后的产物)。
3. Snapshot 格式: `libapp.so` 实际上不仅仅是代码，还包含了一个 Dart VM Snapshot。它没有标准的 ELF 符号表，也没有类似 Java 的类结构元数据。

逆向流程

第 1 步：识别 Flutter 应用

解压 APK，查看 `lib` 目录。如果看到 `libflutter.so` 和 `libapp.so`，那么这肯定是一个 Flutter 应用。

第 2 步：使用 reFlutter 框架

由于 Dart AOT 的特殊性，直接用 IDA 分析 `libapp.so` 非常困难，因为所有函数名都被剥离了，且 Dart 的调用约定和寄存器使用方式与标准 C/C++ 不同。

reFlutter 是目前最强大的 Flutter 逆向辅助工具。它通过修改 Flutter 引擎 (`libflutter.so`)，在应用运行时利用 Dart VM 的内部机制来 Dump 类、函数和偏移信息。

工具: [reFlutter](#)

操作步骤:

1. 重打包: 使用 reFlutter 处理目标 APK。

```
reflutter target.apk
```

1. 安装运行: 安装生成的 `release.RE.apk` 到手机。
2. 获取偏移: 应用启动后, reFlutter 会在 Logcat 中输出关键的 Dart 库函数的偏移地址, 或者生成一个 `dump.dart` 文件。

第 3 步: 流量拦截 (SSL Pinning Bypass)

Flutter 应用不使用系统的代理设置, 也不使用 Java 层的 HTTP 客户端 (OkHttp), 而是使用 Dart 自己的 `HttpClient`。因此, 传统的抓包设置 (Wi-Fi 代理) 和 Frida SSL Pinning 脚本通常无效。

reFlutter 的方案:

reFlutter 在重打包时, 会自动 Patch `libflutter.so` 中的网络校验逻辑, 并强制将流量转发到指定的代理 IP (需要在 reFlutter 配置阶段输入你的 Burp/Charles IP)。这是目前拦截 Flutter 流量最稳定的方法。

Frida 方案 (Hook 验证函数):

如果你不想重打包, 可以使用 Frida Hook `libflutter.so` 中负责验证证书的函数。

- 函数名通常包含 `SessionVerifyCertificateChain`。
- 你需要下载对应 Flutter 版本的 `libflutter.so` 符号文件, 或者通过特征码搜索该函数。
- Hook 该函数并使其直接返回验证成功。

第 4 步：使用 Doldrums 还原代码

Doldrums 是一个针对 Flutter Android 应用的静态分析工具，试图将 `libapp.so` 反编译回 Dart 伪代码。

工具: [Doldrums](#)

注意：由于 Flutter 版本更新极快，Snapshot 格式经常变动，Doldrums 可能不支持最新的 Flutter 版本。

第 5 步：动态分析 (Dart VM Hook)

如果无法静态还原代码，我们需要在运行时进行 Hook。由于没有符号，我们需要结合 reFlutter 导出的偏移地址。

```
// Frida Script Example: Hook Dart Function
// Assume reFlutter tells us the function offset to hook is 0x1a2b3c

var appBase = Module.findBaseAddress("libapp.so");
var targetOffset = 0x1a2b3c;
var targetFunc = appBase.add(targetOffset);

Interceptor.attach(targetFunc, {
    onEnter: function (args) {
        // Dart function parameter passing is special
        // args[0] may not be the first parameter, but a Closure or other VM structure
        // Parameters are usually stored in specific registers or stack locations
        console.log("Dart function called!");

        // Print parameters (try reading first 4 parameters)
        console.log("Arg1: " + args[0]);
        console.log("Arg2: " + args[1]);
        console.log("Arg3: " + args[2]);
    },
    onLeave: function (retval) {
        console.log("Dart function returned: " + retval);
    },
});
```

总结

1. 流量拦截: 必须使用 reFlutter 对 APK 进行 Patch, 或者 Hook `libflutter.so` 中的证书验证函数。
2. 代码分析: 静态分析工具 (如 Doldrums) 兼容性较差, 主要依赖 reFlutter 提取偏移 + Frida 动态调试。
3. 核心: 理解 Dart VM 的工作原理 (Snapshot 结构、Object Pool、Dispatch Table) 是深入逆向 Flutter 的基础。

[C08] 恶意软件分析案例

C08: 安卓银行木马分析案例

恶意软件分析是逆向工程的一个重要应用领域。与常规 App 分析不同，分析恶意软件更关注其隐藏行为、持久化机制、窃密手段以及C2 (Command & Control) 通信。

本案例将模拟分析一个典型的 Android 银行木马 (Banking Trojan)。

样本概况

伪装: 该木马伪装成 "Flash Player" 或 "系统更新" 应用。

行为: 诱导用户开启"无障碍服务"，然后利用该权限进行点击劫持、覆盖攻击 (Overlay Attack)，窃取银行 App 的账号密码，并拦截短信验证码。

详细分析流程

第 1 步：静态分析 (Manifest & 权限)

使用 `jad` 打开 APK，首先查看 `AndroidManifest.xml`。

关键发现:

1. 敏感权限:

- `android.permission.BIND_ACCESSIBILITY_SERVICE` (无障碍服务 - 核心权限)
- `android.permission.RECEIVE_SMS` (接收短信)
- `android.permission.READ_SMS` (读取短信)
- `android.permission.SYSTEM_ALERT_WINDOW` (悬浮窗 - 用于覆盖攻击)

- `android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` (忽略电池优化 - 保活)

2. 入口点 (Entry Points):

- 发现一个 `MainActivity`，但代码很简单，只是请求权限。
- 发现一个 `MyAccessibilityService`，继承自 `AccessibilityService`，这是核心逻辑所在。

第 2 步：分析无障碍服务 (Accessibility Service)

定位到 `MyAccessibilityService` 类，重点分析 `onAccessibilityEvent` 方法。

```
public void onAccessibilityEvent(AccessibilityEvent event) {  
    String packageName = event.getPackageName().toString();  
  
    // 1. Auto grant permissions (Self-Protection & Persistence)  
    if (packageName.equals("com.android.settings")) {  
        // If user opens settings page to uninstall or disable permissions,  
        // malware will auto click "Back" or "Cancel"  
        performGlobalAction(GLOBAL_ACTION_BACK);  
    }  
  
    // 2. Monitor target banking apps  
    if (TARGET_BANK_APPS.contains(packageName)) {  
        // Detected victim opened banking app  
        showOverlay(packageName);  
    }  
  
    // 3. Keyboard recording (Keylogging)  
    if (event.getEventType() == AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED) {  
        String text = event.getText().toString();  
        logKey(text);  
    }  
}
```

第 3 步：覆盖攻击 (Overlay Attack)

窃取：用户以为自己在登录银行，实际上是在木马的 `WebView` 中输入了账号密码。木马通过 `JavaScript Interface` 将输入的数据传回 `Java` 层，然后上传服务器。

第 4 步：短信拦截 (SMS Stealing)

分析 `SmsReceiver` 类。

```
public class SmsReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Object[] pdus = (Object[]) intent.getExtras().get("pdus");
        for (Object pdu : pdus) {
            SmsMessage sms = SmsMessage.createFromPdu((byte[]) pdu);
            String body = sms.getMessageBody();
            String sender = sms.getOriginatingAddress();

            // Upload SMS to C2 Server
            uploadSmsToC2(sender, body);

            // If SMS contains "verification code" or other key words,
            // intercept the broadcast so user cannot see it
            if (is2FACode(body)) {
                abortBroadcast();
            }
        }
    }
}
```

第 5 步：C2 通信分析

1. 定位 C2 地址:

- 资源解密: 有时 C2 地址被加密存储在 `assets` 下的图片或文本文件中，或者使用 DGA (域名生成算法) 动态生成。
- Native 分析: 越来越多的木马将 C2 地址和通信逻辑隐藏在 SO 库中。

2. 通信协议:

- HTTP/HTTPS: 抓包分析 POST 请求。
- WebSocket: 用于实时控制。
- Telegram Bot API: 很多新型木马利用 Telegram Bot 作为 C2，因为 Telegram 的流量通常不会被防火墙拦截，且 HTTPS 难以解密。

Frida Hook 示例 (拦截 Telegram API):

```
// Assume malware uses OkHttp
Java.perform(function () {
    var OkHttpClient = Java.use("okhttp3.OkHttpClient");
    OkHttpClient.newCall.implementation = function (request) {
        var url = request.url().toString();
        if (url.includes("api.telegram.org")) {
            console.log("[!] Detected Telegram C2 Communication: " + url);
        }
        return this.newCall(request);
    };
});
```

第 6 步：反分析技术

木马作者也会使用各种手段防止被逆向：

- 模拟器检测：检查 `Build.FINGERPRINT`, `Build.MODEL` 等。
- 加壳：使用免费或商业的加固服务。
- 动态加载：核心 `dex` 文件被加密存储，运行时动态解密加载 (`DexClassLoader`)。

总结

分析银行木马的关键在于理解其攻击链 (Kill Chain)：

1. Infection: 诱导安装。
2. Persistence: 获取无障碍权限、保活。
3. Stealing: 覆盖攻击窃取凭证、拦截短信。
4. Exfiltration: 将数据回传 C2。

逆向工程师的任务是阻断这一链条，提取 IOC (Indicators of Compromise, 如 C2 域名、文件 Hash)，并协助开发查杀策略。

参考资料

基础知识

[F01] APK 文件结构

F01: APK 文件结构详解

APK (Android Package) 是 Android 操作系统用于分发和安装移动应用的文件格式。它本质上是一个 ZIP 归档文件，包含了应用的所有代码、资源、证书等。理解其内部结构是逆向工程和安全分析的第一步。

目录

1. [APK 概览](#)
2. [AndroidManifest.xml](#)
3. [classes.dex](#)
4. [resources.arsc](#)
5. [res/ 目录](#)
6. [lib/ 目录](#)
7. [assets/ 目录](#)
8. [META-INF/ 目录](#)
9. [APK 分析流程](#)

APK 概览

一个标准的 APK 文件，当用解压缩工具打开时，通常会看到以下目录结构：

```
app.apk
├── AndroidManifest.xml      # [必需] 应用清单文件 (二进制格式)
├── classes.dex              # [必需] Dalvik/ART 字节码
├── classes2.dex             # [可选] 多 DEX 文件
├── resources.arsc            # [必需] 预编译资源索引表
├── res/
│   ├── drawable/             # 图片资源
│   ├── layout/                # 布局 XML
│   ├── values/                # 字符串、颜色等值资源
│   └── ...
├── lib/                      # [可选] 原生库目录
│   ├── arm64-v8a/             # 64位 ARM
│   ├── armeabi-v7a/            # 32位 ARM
│   ├── x86/                   # 32位 x86
│   └── x86_64/                # 64位 x86
├── assets/                   # [可选] 原始资源文件
└── META-INF/
    ├── MANIFEST.MF            # [必需] 签名信息
    ├── CERT.SF
    └── CERT.RSA
```

AndroidManifest.xml

这是 APK 中最重要的文件，它以二进制 XML 格式存储，包含了应用的所有元信息。

核心内容

元素	说明
包名 (Package Name)	应用在系统中的唯一标识符, 如 com.example.app
组件 (Components)	声明所有四大组件: Activity、Service、BroadcastReceiver、ContentProvider
权限 (Permissions)	声明应用需要申请的权限, 如 android.permission.INTERNET
入口点 (Entry Point)	指定哪个 Activity 是应用的启动入口 (LAUNCHER Activity)
SDK 版本	指定 minSdkVersion、targetSdkVersion

四大组件

```

<!-- Activity: 用户界面 -->
<activity android:name=".MainActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<!-- Service: 后台服务 -->
<service android:name=".MyService" android:exported="false" />

<!-- BroadcastReceiver: 广播接收器 -->
<receiver android:name=".MyReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>

<!-- ContentProvider: 内容提供者 -->
<provider android:name=".MyProvider"
          android:authorities="com.example.app.provider"
          android:exported="false" />

```

分析提示：必须使用 `apktool` 或 `jad` 等工具将其解码为人类可读的 XML 格式。直接用文本编辑器打开是乱码。

classes.dex

这是由 Java/Kotlin 代码编译、转换后生成的 Dalvik 虚拟机字节码。应用的所有业务逻辑都在这里。

DEX 文件特点

特性	说明
格式	Dalvik Executable，专为移动设备优化
方法数限制	单个 DEX 文件最多 65536 个方法
多 DEX	超过限制时会有 <code>classes2.dex</code> , <code>classes3.dex</code> 等
字节码	基于寄存器的指令集，比 JVM 字节码更紧凑

分析工具

```
# 使用 jad 反编译为 Java 代码  
jad -d output/ app.apk  
  
# 使用 bksmali 反汇编为 Smali 代码  
bksmali d classes.dex -o smali_output/  
  
# 使用 dex2jar 转换为 JAR  
d2j-dex2jar.sh classes.dex
```

分析提示：使用 `jad` 可以将其反编译为 Java 代码，使用 `bksmali` 可以反汇编为 Smali 代码。

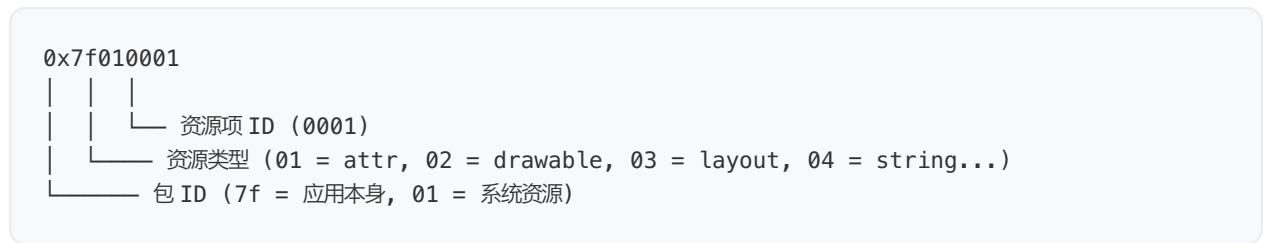
resources.arsc

这是一个二进制资源索引表。Android 系统使用它来快速查找和匹配资源。

功能说明

- 包含从资源 ID 到具体资源文件（或字符串值）的映射关系
- 支持多语言、多屏幕密度等资源适配
- 当代码中调用 `R.string.app_name` 时，系统通过此文件找到对应的字符串值

资源 ID 格式



分析提示: `apktool` 可以解码此文件, 将其中的字符串等资源还原到 `res/values/strings.xml` 等文件中。

res/ 目录

存放未编译的资源文件, 这些资源在打包时大多保持原样或只进行简单处理。

目录结构

目录	内容
res/drawable/	图片资源 (PNG, JPG, WebP, XML drawable)
res/layout/	布局 XML 文件
res/values/	字符串、颜色、尺寸等值资源
res/xml/	任意 XML 配置文件
res/raw/	任意原始二进制文件
res/mipmap/	应用图标资源
res/anim/	动画定义文件

资源限定符

```
res/
└── drawable/          # 默认资源
└── drawable-hdpi/    # 高密度屏幕
└── drawable-xhdpi/   # 超高密度屏幕
└── drawable-xxhdpi/  # 超超高密度屏幕
└── values/            # 默认语言
└── values-zh/         # 中文
└── values-ja/         # 日文
```

分析提示: 布局 XML 文件虽然可读, 但它们引用的字符串等资源是以 `@string/app_name` 的形式存在的, 需要结合 `resources.arsc` 才能完全理解。

lib/ 目录

存放应用使用的 C/C++ 原生库（`.so` 文件）。为了适配不同的 CPU 架构，它通常包含多个子目录。

架构说明

目录	架构	说明
<code>arm64-v8a/</code>	64位 ARM	目前主流 Android 设备
<code>armeabi-v7a/</code>	32位 ARM	较老的 ARM 设备
<code>x86_64/</code>	64位 x86	模拟器、Intel 设备
<code>x86/</code>	32位 x86	模拟器、Intel 设备

常见原生库

```
lib/arm64-v8a/
├── libnative-lib.so      # 应用自定义原生库
├── libflutter.so         # Flutter 引擎
├── libil2cpp.so          # Unity IL2CPP
├── libcrypto.so          # OpenSSL 加密库
└── libsqlcipher.so       # SQLCipher 加密数据库
```

分析提示：核心的加密、解密、核心算法或游戏引擎常常在这里实现。需要使用 IDA Pro、Ghidra、Binary Ninja 等工具进行逆向分析。

assets/ 目录

这是一个“原封不动”的资源目录。与 `res/raw` 类似，这里的任何文件在打包时都不会被系统处理。

常见用途

类型	示例
配置文件	<code>config.json</code> , <code>settings.xml</code>
Web 资源	<code>index.html</code> , <code>app.js</code> , <code>style.css</code>
游戏资源	地图数据、关卡配置、精灵图
字体文件	<code>custom_font.ttf</code>
数据库	预置的 SQLite 数据库
加密数据	加密的配置或密钥文件

分析提示: 检查此目录是否有敏感的配置文件、密钥或 Web 资源。

META-INF/ 目录

存放应用的签名信息，用于验证 APK 的完整性和来源。

文件说明

文件	说明
<code>MANIFEST.MF</code>	包含 APK 中每个文件的名称及其 SHA-256 哈希值
<code>CERT.SF</code>	APK 中每个文件的摘要（哈希值）
<code>CERT.RSA</code>	包含用于签署 <code>CERT.SF</code> 的公钥和证书

签名机制

1. 计算 APK 中每个文件的哈希 → 记录在 MANIFEST.MF
2. 计算 MANIFEST.MF 整个文件的哈希 → 记录在 CERT.SF
3. 用开发者私钥对 CERT.SF 签名 → 生成 CERT.RSA

当系统安装 APK 时，会用 CERT.RSA 中的公钥来验证签名，确保文件自签名后未被篡改。

APK 签名方案

版本	说明
v1 (JAR 签名)	传统签名方式，基于 META-INF 目录
v2 (APK 签名块)	Android 7.0+，签名信息存储在 APK 签名块中
v3 (APK 签名块 + 密钥轮换)	Android 9.0+，支持密钥轮换
v4 (增量签名)	Android 11+，支持增量安装

分析提示：对 APK 进行任何修改（包括重打包）后，都必须用自己的密钥重新签名，否则安装会失败。

APK 分析流程

1. 解包

```
# 推荐: 使用 apktool 解包 (正确解码 XML 和资源)  
apktool d app.apk -o app_decoded/  
  
# 备选: 直接解压 (不解码二进制 XML)  
unzip app.apk -d app_unzipped/
```

2. 静态分析

```
# 使用 jadx-gui 打开, 直接浏览反编译的 Java 代码  
jadx-gui app.apk  
  
# 命令行批量反编译  
jadx -d java_output/ app.apk
```

检查清单:

- 阅读 `AndroidManifest.xml`, 了解主要组件、权限和入口点
- 检查 `assets/` 目录, 寻找配置文件或敏感数据
- 检查 `lib/` 目录, 识别使用的原生库
- 搜索硬编码的密钥、URL、API 端点

3. 动态分析

```
# 安装应用  
adb install app.apk  
  
# 使用 Frida 进行 Hook  
frida -U -f com.example.app -l hook_script.js  
  
# 使用 objection 快速分析  
objection -g com.example.app explore
```

4. 重打包（修改后）

```
# 使用 apktool 重新打包  
apktool b app_decoded/ -o app_modified.apk  
  
# 签名  
apksigner sign --ks my.keystore app_modified.apk  
  
# 或使用 jarsigner  
jarsigner -keystore my.keystore app_modified.apk alias_name  
  
# 对齐优化  
zipalign -v 4 app_modified.apk app_aligned.apk
```

总结

文件/目录	分析重点
AndroidManifest.xml	组件、权限、入口点、导出状态
classes.dex	业务逻辑、加密算法、网络请求
lib/*.so	核心算法、加密实现、反调试
assets/	配置文件、密钥、Web 资源
META-INF/	签名信息、证书

掌握 APK 文件结构是 Android 逆向工程的基础。通过系统地分析每个组成部分，可以全面了解应用的功能和实现细节。

[F02] Android 四大组件

F02: Android 四大组件

Android 的应用框架核心由四个基本组件构成。每个组件都是一个独立的实体，系统和应用可以通过它进入你的 App。理解这四个组件的职责和生命周期是进行任何 Android 开发或逆向分析的基础。

1. 活动 (Activity)

- 概念: Activity 是用户界面的单一屏幕。它为用户提供了一个可以进行交互的操作界面。一个 App 通常由多个相互关联的 Activity 组成。
- 核心职责:
 - UI 承载: 负责绘制用户界面、承载 `View` 和 `ViewGroup`。
 - 用户交互: 响应用户的点击、滑动、输入等事件。
 - 生命周期管理: 管理从创建到销毁的整个生命周期, 以响应系统状态的变化 (如来电、屏幕旋转)。
- 生命周期:

一个 Activity 具有清晰的生命周期回调方法, 这对于逆向分析至关重要, 因为核心逻辑 (如数据加载、UI 更新) 常常在这些方法中被触发。

 - `onCreate()`: Activity 被创建。这是最重要的回调, 通常在这里进行布局加载 (`setContentView`)、数据初始化、事件绑定等。
 - `onStart()`: Activity 变得可见, 但还不能与用户交互。

- `onResume()`: Activity 到达前台，可以与用户进行交互。这是 Hook UI 相关逻辑的绝佳位置。
 - `onPause()`: Activity 即将进入后台，不再是焦点。通常在这里保存未提交的数据。
 - `onStop()`: Activity 完全不可见。
 - `onDestroy()`: Activity 即将被销毁。
 - `onRestart()`: Activity 从停止状态重新启动。
- 逆向切入点:
 - Hook `onCreate()` 或 `onResume()` 是分析一个新页面的标准起点。
 - 通过 `adb shell dumpsys activity top` 可以查看当前位于前台的 Activity 的类名，这是快速定位目标页面的关键命令。

2. 服务 (Service)

- 概念: Service 是一个在后台执行长时间运行操作而没有用户界面的组件。即使用户切换到其他应用，服务仍然可以继续工作。
- 核心职责:
 - 后台任务: 执行不需要 UI 的任务，如播放音乐、下载文件、同步数据。
 - 进程间通信 (IPC): 可以作为服务端，为其他 App 提供功能。
- 类型:
 - 启动服务 (Started Service): 通过 `startService()` 启动，一旦启动，服务就可以无限期地在后台运行，直到它自己停止或被系统销毁。
 - 绑定服务 (Bound Service): 通过 `bindService()` 启动。它提供了一个客户端-服务器接口，允许组件（如 Activity）与服务进行交互、发送请求、获取结果。当所有绑定的组件都解绑后，服务就会被销毁。

- 前台服务 (Foreground Service): 为了防止被系统轻易杀死, Service 可以通过 `startForeground()` 将自己提升为前台服务, 此时必须在状态栏显示一个持续的通知 (例如音乐播放通知)。
- 逆向切入点:
 - 很多 App 的核心业务逻辑 (如消息推送、位置上报、数据同步) 都放在 Service 中。
 - Hook Service 的 `onStartCommand()` 或 `onBind()` 方法可以帮助理解其后台行为。

3. 广播接收器 (Broadcast Receiver)

- 概念: 广播接收器是一个用于响应系统范围广播通知的组件。许多广播源自系统 (例如, 屏幕关闭、网络状态改变、电池电量低), 但应用也可以发起自定义广播。
- 核心职责:
 - 监听系统事件: 让 App 能够对设备状态的变化做出反应。
 - 应用间通信: 一个 App 可以向其他 App 发送广播, 实现简单的消息通知。
- 类型:
 - 静态注册: 在 `AndroidManifest.xml` 中使用 `<receiver>` 标签声明。即使 App 没有运行, 当广播事件发生时, 系统也会唤醒 App 来处理它。
 - 动态注册: 在代码中通过 `Context.registerReceiver()` 注册。它的生命周期与注册它的组件 (如 Activity) 相关联。
- 逆向切入点:
 - 分析 `AndroidManifest.xml` 中的静态广播接收器, 可以了解 App 关心哪些系统事件。
 - Hook `onReceive()` 方法是捕获和分析广播内容 (Intent) 的关键。

4. 内容提供器 (Content Provider)

- 概念: 内容提供器用于管理一组共享的应用数据。它以一种标准化的接口，将数据暴露给其他应用。数据可以存储在文件系统、SQLite 数据库或任何其他持久化存储位置。
- 核心职责:
 - 数据共享: 提供一个安全、统一的接口，让其他应用可以查询或修改本应用的数据。
 - 数据抽象: 隐藏了底层数据的存储细节。无论数据是存在数据库还是文件中，对外的接口都是一致的。
 - 权限控制: 可以精细地控制其他应用对数据的读写权限。
- 工作方式:
 - 通过一个唯一的 `URI` (Uniform Resource Identifier) 来标识数据。例如
`content://com.example.app.provider/users/10`。
 - 其他应用使用 `ContentResolver` 对象，通过 `query()`, `insert()`, `update()`,
`delete()` 等方法与 Content Provider 进行交互。
- 逆向切入点:
 - App 的联系人、短信、媒体库等都是通过 Content Provider 访问的。
 - 分析 `AndroidManifest.xml` 中声明的 `provider`，可以找到 App 对外暴露了哪些数据。
 - 逆向 App 时，可以自己编写一个 App 来调用目标 App 的 Content Provider，从而读取或操纵其内部数据。

[F03] AndroidManifest 解析

F03: AndroidManifest.xml 深度解析

`AndroidManifest.xml` 是 Android 应用的"大脑"和"蓝图"。它是一个强制性的配置文件，位于每个 APK 的根目录中。该文件向 Android 构建工具、操作系统和 Google Play 描述了应用的基本信息、组件、权限和硬件要求。对于逆向工程师来说，这是了解应用功能、入口点和安全边界的主要切入点。

核心作用与特性

- 唯一标识: 定义了应用的 Java 包名，这是它在设备和 Google Play 上的唯一标识。
- 组件声明: 声明应用的所有核心组件（四大组件）。任何未在此文件中声明的组件都对系统不可见，也无法运行。
- 权限请求: 列出应用需要访问的受保护部分 API 或系统资源所需的权限。
- 硬件/软件要求: 声明应用运行所需的硬件功能（如摄像头、蓝牙）和最低 Android API 级别。
- 入口点定义: 通过 `intent-filter` 指定哪个 Activity 是应用的启动器。
- 重要提示: 原始的 `AndroidManifest.xml` 是二进制格式的。必须使用 `apktool`, `jad`, `aapt` 等工具解码后才能阅读。

关键标签 (Tags) 详解

`<manifest>`

根元素。它必须包含 `package` 属性来定义应用的唯一包名。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.myapp">  
    ...  
</manifest>
```

- `android:theme` : 应用的全局主题。
- `android:name` : 指定 `Application` 子类的名称, 常用于应用初始化。这是 Hook 的绝佳目标。
- `android:debuggable` : (安全关键) `true` 表示应用是可调试的, 允许 `adb` 连接和任意代码执行。发布的 Release 版本必须为 `false`。
- `android:allowBackup` : (安全关键) `true` 允许用户通过 `adb backup` 备份应用数据。如果应用数据敏感, 应设为 `false`。
- `android:networkSecurityConfig` : (安全关键) 指向网络安全配置文件, 用于定义 SSL Pinning、自定义 CA 等高级网络策略。

<activity>

声明一个 Activity (UI 界面)。

- `android:name` : Activity 类的名称。`.MyActivity` 是 `package.MyActivity` 的简写。
- `android:exported` : (安全关键) `true` 表示该 Activity 可以被其他应用启动。如果该 Activity 处理敏感数据且无需外部调用, 应设为 `false`, 否则可能导致组件劫持和数据泄露。对于包含 `LAUNCHER` intent-filter 的 Activity, `exported` 默认为 `true`。

<service>

声明一个 Service (后台服务)。

- `android:name` : Service 类的名称。
- `android:exported` : (安全关键) `true` 表示该 Service 可以被其他应用绑定或启动。规则同 Activity。

<receiver>

声明一个 BroadcastReceiver (广播接收器)。

- `android:name` : Receiver 类的名称。
- `android:exported` : (安全关键) `true` 表示它可以接收来自系统或其他应用的广播。

<provider>

声明一个 ContentProvider (内容提供者), 用于跨应用共享数据。

- `android:name` : Provider 类的名称。
- `android:authorities` : Provider 的唯一标识符, 通常是包名加上描述性后缀。
- `android:exported` : (安全关键) `true` 表示其他应用可以访问其数据。如果 `minSdkVersion` 或 `targetSdkVersion` ≥ 17 , 默认值为 `false`。不正确的 `exported` 设置可能导致 SQL 注入或文件遍历漏洞。

<uses-permission>

请求应用运行所需的权限。这是分析应用行为的关键。

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.CAMERA" />
```

1. **确定入口点**:

- * 寻找 `MAIN/LAUNCHER` 的 Activity。
- * 寻找 `android:name` 属性定义的 `Application` 子类，这是最早执行代码的地方。

2. **识别核心功能**:

- * 阅读权限列表 (`<uses-permission>`），快速了解应用能力。

- * 查看声明的 Activities、Services，推测其功能模块。

3. **寻找攻击面**:

- * 检查所有组件的 `android:exported="true"` 属性，这些是潜在的攻击入口。

- * 分析 `intent-filter`，特别是自定义的 `scheme`，寻找 URL Scheme 漏洞。

- * 检查 `android:debuggable="true"`，如果为 `true`，可以直接附加调试器。

- * 检查 `android:allowBackup="true"`，尝试 `adb backup` 导出数据。

安全风险与配置

- * **组件导出风险**: 错误地将内部组件设置为 `exported="true"` 是最常见的 Android 漏洞之一。

- * **Webview 风险**: 检查是否使用了 `WebView`，并确认是否开启了 `setJavaScriptEnabled(true)`，这可能导致远程代码执行。

- * **File Provider 路径遍历**: 如果 `ContentProvider` 是 `FileProvider`，不正确的配置可能导致任意文件读取。

- * **硬编码密钥**: 虽然不在 Manifest 中，但分析后应在 `res/values/strings.xml` 或代码中寻找硬编码的 API 密钥或 URL。

[F04] Android Studio 调试工具

F04: Android Studio 调试工具集

Android Studio 不仅仅是一个代码编辑器，它还集成了一套强大、可视化的调试和分析工具，能够极大地提升开发和逆向分析的效率。熟悉这些工具是每个 Android 工程师的必备技能。

1. 日志猫 (Logcat)

- 概念: Logcat 是一个命令行工具，用于查看 Android 系统的日志输出。Android Studio 将其封装在一个便捷的窗口中，可以实时查看来自系统和所有应用的日志信息。
- 核心功能:
 - 实时日志流: 显示来自设备或模拟器的连续日志。
 - 日志级别过滤: 可以按 Verbose, Debug, Info, Warn, Error, Assert 等级别进行过滤。在调试时，通常关注 Error 和 Warn 级别。
 - 进程/应用过滤: 可以只显示当前正在调试的应用，或特定进程的日志。
 - 关键词搜索: 强大的搜索功能，支持正则表达式，可以快速找到你关心的日志条目。
 - 堆栈跟踪点击: 当日志中包含异常堆栈时，可以直接点击类名和行号，跳转到源码的对应位置。
- 逆向应用:
 - 很多 App 在开发阶段会留下大量的调试日志，其中可能包含敏感信息，如加密前的请求参数、服务器返回的明文等。
 - 通过观察 App 在特定操作下的日志输出，可以推断其内部的工作流程。

2. 布局检查器 (Layout Inspector)

- 概念: 这是一个强大的可视化工具, 允许你实时检查和调试正在运行的应用的视图层次结构。
- 核心功能:
 - 3D 视图层次: 以 3D 可旋转的视图展示当前 Activity 的所有 `View` 组件是如何嵌套的。
 - 属性查看: 点击任何一个 `View`, 都可以实时查看其所有的属性, 如 `ID`, `text`, `width`, `height`, `padding`, `color` 等。
 - 实时更新: 在设备上操作 UI, 布局检查器中的视图会实时同步更新。
 - 定位重叠视图: 可以轻松发现被其他视图遮挡或尺寸为 0 的不可见视图。
- 逆向应用:
 - 快速定位一个按钮或文本框的 `resource-id`, 这是编写 UIAutomator2 等自动化脚本的关键。
 - 分析自定义 `View` 的绘制方式和内部结构。
 - 理解一些反截图、反录屏技术是如何通过添加一个透明的 `FLAG_SECURE` 窗口来实现的。

3. 应用分析器 (Profiler)

- 概念: Profiler 是一套用于实时分析应用性能的工具, 主要关注 CPU、内存、网络和电量四个方面。
- 核心功能:
 - CPU Profiler:
 - 方法追踪 (Method Tracing): 可以记录下在一个时间段内所有函数的调用栈和执行时间, 生成火焰图。这对于寻找性能瓶颈至关重要。
 - 系统调用追踪: 可以查看 App 与系统内核的交互情况。

-
- Memory Profiler:
 - 实时内存占用: 显示 App 的 Java 堆、Native 堆和图形内存的实时使用情况。
 - 堆转储 (Heap Dump): 可以捕获某一时刻的内存快照，并分析其中所有的对象实例、引用关系。
 - 内存泄漏检测: 长期运行并观察内存曲线，可以帮助定位内存泄漏问题。
 - Network Profiler:
 - 网络请求时间线: 以时间线的形式展示 App 发起的所有网络请求，包括请求的建立时间、发送时间和接收时间。
 - 请求详情: 可以查看每个请求的 URL、请求头、响应头和 Payload。（注意：对于 HTTPS 流量，默认只能看到加密后的内容，需要配合其他抓包工具使用）。
 - 逆向应用:
 - 方法追踪: 通过对一个加密函数进行追踪，可以完整地了解其内部都调用了哪些子函数，帮助理解复杂算法。
 - 堆转储: 在 App 解密并加载了某些核心数据到内存后，通过 Heap Dump 可能会找到这些数据对象，甚至是密钥对象。
 - 网络分析: 虽然不能解密 HTTPS，但可以快速了解 App 在何时、向哪些域名发起了网络请求，为网络抓包提供目标。

4. 数据库检查器 (Database Inspector)

- 概念: 允许你实时检查、查询和修改正在运行的应用中的数据库 (SQLite)。
- 核心功能:
 - 实时数据查看: 实时显示数据库中所有表的内容。
 - 自定义查询: 可以直接运行 SQL 查询语句来检索或修改数据。
 - 数据修改: 可以直接在图形界面中修改字段的值。
 - 保持连接: 即使 App 进程被杀死后重启，检查器也能保持连接。

- 逆向应用:

- 很多 App 会将用户信息、配置、聊天记录等重要数据存储在 SQLite 数据库中。通过数据库检查器，可以一目了然地分析其数据结构和内容。
 - 通过修改数据库中的值（例如，将会员状态字段从 0 改为 1），可以测试和验证一些本地的 VIP 功能绕过。
-

[F05] DEX 文件格式

F05: DEX 文件权威指南

DEX (Dalvik Executable) 文件是 Android 操作系统的核心组成部分之一。它们是专门为在内存和处理器速度受限的设备上高效运行而设计的。本指南将深入探讨 DEX 文件的定义、格式、运行原理以及相关工具。

!!! question "思考：理解 DEX 格式的实战价值" 很多初学者会问："DEX 格式这么复杂，我真的需要了解这些底层细节吗？"

考虑这些实际场景：

- 加固对抗：当 App 使用了 DEX 加壳（如梆梆、360），你需要知道 DEX 的魔数、签名字段在哪，才能判断脱壳是否完整
- 动态加载分析：很多 App 会在运行时解密并加载隐藏的 DEX，理解 `Class Defs` 结构能帮你快速定位被隐藏的恶意代码
- Multi-DEX 定位：当你想 Hook 某个类，但不知道它在哪个 `classes.dex` 中时，理解 String IDs 和 Type IDs 能帮你快速搜索
- 方法数优化：理解 65536 方法数限制的根本原因（Method IDs 索引用 16 位），能帮你更好地进行模块化设计

DEX 格式不是学术知识，而是你破解加固、分析恶意代码的手术刀。

目录

1. 定义与角色：[什么是 DEX 文件？](#)
2. DEX vs. CLASS：[与 Java 字节码的对比](#)
3. DEX 文件结构：[深入剖析格式](#)

[4. 运行原理: DEX 文件如何被执行?](#)[5. Multi-DEX: 应对方法数限制](#)[6. DEX 分析与处理工具](#)

定义与角色

DEX 文件是包含了 Android 应用代码的单个可执行文件。在打包 (Build) 过程中, Java 编译器首先将 `.java` 源码文件编译成标准的 Java 字节码 `.class` 文件。然后, Android SDK 中的 `d8` 工具 (旧版本为 `dx`) 会将所有的 `.class` 文件 (包括项目代码和依赖库) 优化并合并成一个或多个 `classes.dex` 文件。

这个 `classes.dex` 文件最终被打包进 APK (Android Package) 中。当用户安装并运行应用时, Android 系统 (特别是 ART) 会直接执行 DEX 文件中的代码。

核心角色:

- **紧凑性:** 将所有类文件合并, 并共享字符串和常量, 大大减少了文件体积和 I/O 开销。
- **高效性:** 采用基于寄存器的指令集, 更接近底层硬件, 执行效率比基于栈的 JVM 更高。
- **移动优化:** 专为内存有限的移动设备设计。

本文档参考了 Android 官方关于 [DEX 文件格式](#) 的说明。

DEX vs. CLASS

特性	<code>.class</code> 文件 (JVM)	<code>.dex</code> 文件 (ART/Dalvik)
文件数量	每个源文件对应一个 <code>.class</code> 文件	所有 <code>.class</code> 文件合并成一个或多个 <code>.dex</code> 文件
指令集架构	基于栈 (Stack-based)	基于寄存器 (Register-based)
常量池	每个文件都有自己独立的常量池	所有类共享一个全局的字符串和常量池
冗余信息	大量冗余字符串 (如类名、方法名)	字符串和常量去重，通过索引引用，冗余少
平台	任何有 JVM 的地方	Android 平台
转换工具	<code>javac</code>	<code>javac -> d8 / dx</code>

DEX 文件结构

DEX 文件格式非常紧凑和高效，其结构可以大致分为以下几个部分，并由一个 `header` 来描述整个文件的元数据和偏移量。

!!! tip "逆向技巧：从结构入手快速定位" 面对一个陌生的 DEX 文件，如何快速找到你感兴趣的代码？

自顶向下的分析策略：

1. 看 Header：检查魔数确认文件完整性，查看 `class_defs_size` 了解有多少个类

-
2. 搜 String IDs: 用 `dexdump` 或 `strings` 搜索关键字符串（如 "encrypt", "http://"），定位可疑代码
 3. 查 Method IDs: 通过方法名索引找到具体实现
 4. 跳 Class Defs: 直接定位到目标类的完整定义
 5. 读 Code Item: 最后才深入字节码细节

这种"线索驱动"的方法，比漫无目的地浏览代码高效得多。

A DEX file consists of several main sections:

1. 头部 (Header)

- Header: 文件头，包含魔数（`dex\n035\0`）、校验和、签名，以及指向其他数据结构（如字符串、类定义等）的偏移量和大小。
- String IDs: 字符串标识符列表。包含 DEX 文件中用到的所有字符串（如类名、方法名、变量名、字符串常量），并为每个字符串分配一个唯一的 ID。
- Type IDs: 类型标识符列表。包含代码中用到的所有类型（类、接口、数组、基本类型），并指向 `String IDs` 中的相应字符串。
- Proto IDs: 方法原型标识符列表。定义了方法的返回类型和参数类型。
- Field IDs: 字段标识符列表。定义了类的成员变量，包括其所属类、类型和名称。
- Method IDs: 方法标识符列表。定义了方法，包括其所属类、原型 (Proto ID) 和名称。
- Class Defs: 类定义列表。这是核心部分，包含了每个类的详细信息：访问标志、父类、实现的接口、源码文件名、注解、以及指向其字段和方法的指针。
- Data Section: 数据区，包含了所有类的实际内容，例如：
- Code Item: 实际的方法字节码 (Dalvik 指令)。
- Class Data: 类的字段和方法列表的具体数据。
- Map List: 描述整个 DEX 文件数据布局的映射表，`dexdump` 等工具使用它来解析文件。

运行原理

DEX 文件的执行由 Android 运行时 (ART) 负责，在 Android 5.0 之前由 Dalvik 虚拟机 (DVM) 负责。

1. Dalvik 虚拟机 (DVM) - android 4.4 及更早版本

- JIT (Just-In-Time) 编译: 当应用运行时, DVM 会解释执行 DEX 字节码。对于频繁执行的"热点"代码路径, JIT 编译器会将其动态编译成本地机器码, 以提高后续执行速度。
- 缺点: 每次启动应用都需要进行解释和 JIT 编译, 导致应用启动速度较慢, 且运行时消耗更多计算资源。

2. android 运行时 (ART) - Android 5.0 及更高版本

- AOT (Ahead-Of-Time) 编译: 在应用安装时, ART 会使用 `dex2oat` 工具将 DEX 文件中的字节码预编译成设备原生的机器码, 并保存为 OAT (Optimized Android file format) 文件。
- 优点:
 - 启动速度快: 应用直接执行预编译的本地代码, 无需实时编译, 大大加快了启动速度。
 - 性能更高: AOT 可以进行更深度的优化, 性能通常优于 JIT。
 - 更省电: 减少了运行时的 CPU 计算负担。
- AOT + JIT 混合模式 (Android 7.0+):
 - 为了平衡安装速度/空间占用和性能, ART 引入了混合模式。
 - 安装时: 不进行完全 AOT 编译, 或只编译部分关键代码。
 - 首次运行: 解释执行, 并使用 JIT 编译热点代码, 同时收集分析信息 (Profile)。
 - 设备空闲时: 当设备充电且空闲时, 系统会根据收集到的分析信息, 对常用代码进行 AOT 编译, 实现最佳性能。

Multi-DEX

单个 DEX 文件有一个方法引用数上限（65,536 个），当应用（包括其依赖库）的方法总数超过这个限制时，编译会失败。

为了解决这个问题，Android 引入了 Multi-DEX 机制。打包工具会将应用代码分割到多个 DEX 文件中，例如 `classes.dex`，`classes2.dex`，`classes3.dex` 等。主 `classes.dex` 文件会优先加载，然后应用代码会负责加载其余的 DEX 文件。

从 Android 5.0 (API 21) 开始，ART 原生支持加载多个 DEX 文件，无需额外的库。对于更早的版本，则需要使用官方的 [multidex-support-library](#)。

DEX 分析与处理工具

工具	描述
d8 / dx	Google 官方工具，用于将 <code>.class</code> 文件转换为 <code>.dex</code> 文件。 <code>d8</code> 是新一代的转换器。
dexdump	位于 Android SDK <code>build-tools</code> 中，用于打印 DEX 文件的详细信息，包括头信息、类、方法和字节码。
baksmali	将 <code>.dex</code> 文件反汇编成 <code>.smali</code> 文件。Smali 是一种人类可读的 Dalvik 字节码表示形式。
smali	将 <code>.smali</code> 文件重新汇编成 <code>.dex</code> 文件。常用于修改应用逻辑后重新打包。
Jadx	非常强大的反编译工具，可以直接将 APK/DEX 文件反编译成可读的 Java 代码，并提供图形化界面。
Ghidra / IDA Pro	高级逆向工程工具，支持对 DEX 文件和原生库进行深度静态和动态分析。

[F06] Smali 语法入门

F06: Smali 语法入门指南

Smali/Baksmali 是 Dalvik 虚拟机字节码的汇编器/反汇编器。Smali 是对 DEX 格式的一种人类可读的表示，允许我们精确地查看和修改应用的行为。理解 Smali 是进行 Android 应用静态 patching（修改后重打包）的关键。

目录

1. 基本概念
2. 数据类型与表示
3. 文件与类结构
4. 常用指令
5. Smali 实战：修改方法

基本概念

寄存器 (Registers)

Dalvik VM 是基于寄存器的。方法内的局部变量存储在寄存器中。

寄存器类型	说明
v0 , v1 , v2 ...	本地变量寄存器
p0 , p1 , p2 ...	方法参数寄存器

关于 `p0` 的特殊说明：

- 对于非静态方法，`p0` 总是指向 `this`（当前对象实例），参数从 `p1` 开始
- 对于静态方法，参数从 `p0` 开始

示例：一个有两个参数的非静态方法：

- `p0` = `this`
- `p1` = 第一个参数
- `p2` = 第二个参数

常用声明

声明	说明
<code>.locals</code>	声明方法使用的本地变量寄存器数量
<code>.prologue</code>	方法体的序言部分
<code>.line</code>	对应原始 Java 代码的行号，用于调试

数据类型与表示

Smali 使用特定的描述符来表示 Java 中的数据类型。

Smali 类型	Java 类型	描述
V	void	空返回类型
Z	boolean	布尔值
B	byte	字节
S	short	短整型
C	char	字符
I	int	整型
J	long	长整型 (占用两个寄存器)
F	float	浮点型
D	double	双精度浮点型 (占用两个寄存器)
L<包名>/<类名>;	package.ClassName	对象类型, 以 L 开头, ; 结尾
[<类型>]	type[]	数组类型

类型表示示例

Smali	Java
Ljava/lang/String;	java.lang.String
[I	int []
[[Ljava/lang/Object;	java.lang.Object[][]
[Ljava/lang/String;	String []

文件与类结构

每个 `.smali` 文件对应一个 Java 类。

类定义

```
# 定义类、访问修饰符和完整类路径
.class public Lcom/example/app/MainActivity;

# 定义父类
.super Landroid/app/Activity;

# 定义源文件名（可选）
.source "MainActivity.java"
```

字段定义

```
# 格式: .field <访问修饰符> [static] [final] <字段名>:<字段类型>

# 实例字段
.field private TAG:Ljava/lang/String;

# 静态常量字段
.field public static final MY_CONSTANT:I = 0x1
```

方法定义

```
# 格式: .method <访问修饰符> [static] [final] <方法名>(<参数类型>) <返回类型>
.method public onCreate(Landroid/os/Bundle;)V
# 声明本地变量寄存器数量
.locals 3

# 声明参数寄存器
.param p1, "savedInstanceState"    # p1 是 savedInstanceState

# 方法体开始
.prologue
.line 15

# ... Smali 指令 ...

# 方法返回
return-void
.end method
```

常用指令

赋值与移动指令

指令	说明
const-string v1, "Hello"	将字符串 "Hello" 赋值给 v1
const/4 v0, 0x1	将 4 位常量 1 赋值给 v0
const/16 v0, 0x100	将 16 位常量赋值给 v0
move-result-object v0	将上一个 invoke 返回的对象结果移动到 v0
move-result v0	将上一个 invoke 返回的非对象结果移动到 v0
move-exception v0	在 catch 块中, 将捕获的异常对象移动到 v0

对象操作指令

指令	说明
<code>new-instance v0, Ljava/lang/StringBuilder;</code>	创建新实例
<code>iget-object v0, p0, Lcom/example/MyClass;->myField:Ljava/lang/String;</code>	获取实例字段
<code>iput-object v1, p0, Lcom/example/MyClass;->myField:Ljava/lang/String;</code>	设置实例字段
<code>sget-object v0, Lcom/example/Constants;->SOME_STRING:Ljava/lang/String;</code>	获取静态字段
<code>sput-object v0, Lcom/example/Constants;->SOME_STRING:Ljava/lang/String;</code>	设置静态字段

方法调用指令

指令	说明
<code>invoke-virtual {p0, p1}, Lcom/example/MyClass;->myMethod(I)V</code>	调用虚方法（公有/保护方法）
<code>invoke-direct {p0}, Ljava/lang/Object;-><init>()V</code>	调用直接方法（私有方法或构造函数）
<code>invoke-static {v0}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I</code>	调用静态方法
<code>invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V</code>	调用父类方法
<code>invoke-interface {p0, v0}, Ljava/util/List;->add(Ljava/lang/Object;)Z</code>	调用接口方法

跳转/条件指令

指令	说明
<code>goto :label_10</code>	无条件跳转到 <code>:label_10</code>
<code>if-eqz v0, :label_10</code>	如果 <code>v0 == 0</code> (或 <code>false / null</code>) , 则跳转
<code>if-nez v0, :label_10</code>	如果 <code>v0 != 0</code> (或 <code>true / not null</code>) , 则跳转
<code>if-eq v0, v1, :label_10</code>	如果 <code>v0 == v1</code> , 则跳转
<code>if-ne v0, v1, :label_10</code>	如果 <code>v0 != v1</code> , 则跳转
<code>if-lt v0, v1, :label_10</code>	如果 <code>v0 < v1</code> , 则跳转
<code>if-ge v0, v1, :label_10</code>	如果 <code>v0 >= v1</code> , 则跳转

运算指令

指令	说明
<code>add-int v0, v1, v2</code>	<code>v0 = v1 + v2</code> (整型加法)
<code>sub-int v0, v1, v2</code>	<code>v0 = v1 - v2</code> (整型减法)
<code>mul-int v0, v1, v2</code>	<code>v0 = v1 * v2</code> (整型乘法)
<code>div-int v0, v1, v2</code>	<code>v0 = v1 / v2</code> (整型除法)
<code>mul-int/2addr v0, v1</code>	<code>v0 = v0 * v1</code> (结果存回第一个寄存器)

Smali 实战：修改方法

假设我们要修改一个方法，让它总是返回 `true`。

原始 Java 代码

```
public class LicenseCheck {  
    public boolean isLicensed() {  
        // ... 复杂的检查逻辑 ...  
        return false;  
    }  
}
```

原始 Smali 代码

```
.method public isLicensed()Z  
.locals 1  
  
# ... 复杂检查逻辑对应的 smali 指令 ...  
  
const/4 v0, 0x0      # v0 = 0 (false)  
return v0  
.end method
```

修改后的 Smali 代码

```
.method public isLicensed()Z  
.locals 1  
  
# 删除所有复杂检查逻辑，直接返回 true  
  
const/4 v0, 0x1      # v0 = 1 (true)  
return v0  
.end method
```

修改步骤

1. 使用 `apktool d app.apk -o app_decoded/` 解包 APK

2. 找到并修改目标 `.smali` 文件
3. 使用 `apktool b app_decoded/ -o new_app.apk` 重新打包
4. 用 `jarsigner` 或 `apksigner` 对 `new_app.apk` 进行签名

常见修改技巧

1. 让方法返回固定值

```
# 返回 true  
const/4 v0, 0x1  
return v0  
  
# 返回 false  
const/4 v0, 0x0  
return v0  
  
# 返回 null  
const/4 v0, 0x0  
return-object v0
```

2. 跳过方法体直接返回

```
.method public checkSomething()V  
.locals 0  
  
# 直接返回, 跳过所有检查  
return-void  
.end method
```

3. 修改条件判断

```
# 原始: if-eqz v0, :skip (如果 v0 == 0 则跳过)  
# 修改: if-nez v0, :skip (如果 v0 != 0 则跳过)  
# 或者直接: goto :skip (无条件跳过)
```

[F07] SO/ELF 文件格式

F07: Android .so 文件详解 (ELF Format)

.so 文件 (Shared Object) 是 Android 平台上的原生共享库，等同于 Windows 上的 .dll 或 Linux 上的 .so。它们包含了由 C/C++ 等原生代码编译而成的机器码。在 Android 逆向工程中，分析 .so 文件是理解应用核心逻辑、破解加密算法和绕过安全机制的关键一步。

目录

1. ELF 文件格式

- ELF Header
- Program Header Table
- Section Header Table
- 关键 Section

2. 加载与链接

- System.loadLibrary()
- JNI (Java Native Interface)
- 动态链接器

3. 静态分析

- 识别关键函数
- 使用 IDA Pro / Ghidra

4. 动态分析

- Frida Hook 原生函数
- Unidbg 模拟执行

5. 常见保护手段

- 字符串加密

- 代码混淆

- 反调试技术

6. init_array 详解

- 调用时机

- 逆向分析对策

ELF 文件格式

.so 文件遵循 ELF (Executable and Linkable Format) 格式，这是一种用于可执行文件、目标代码、共享库和核心转储的标准文件格式。

ELF Header

位于文件开头，描述了整个文件的"档案"，包括：

字段	说明
Magic Number	文件的前 16 个字节，用于识别这是一个 ELF 文件
Architecture	标识文件是为哪种 CPU 架构编译的（如 ARM, ARM64, x86）
Type	文件类型（可执行文件、共享库等）
Entry Point Address	如果是可执行文件，这是程序启动的地址
Program Header Table Offset	指向程序头表的偏移
Section Header Table Offset	指向节头表的偏移

Program Header Table

描述了系统如何将文件的各个部分（段，Segments）加载到内存中。每个条目都定义了一个段的类型（如 `LOAD`，表示需要加载到内存）、虚拟地址、物理地址、大小和权限（读、写、执行）。动态链接器（linker）依赖这个表来正确映射 `.so` 文件。

Section Header Table

描述了文件中各个"节"（Sections）的信息。节是链接器用来组织和处理数据的单位。

关键 Section

Section	说明
<code>.text</code>	包含已编译的程序机器码（汇编指令）。这是分析的核心区域
<code>.rodata</code>	只读数据区，通常存放字符串常量、const 变量等
<code>.data</code>	已初始化的可读可写数据区（全局变量和静态变量）
<code>.bss</code>	未初始化的数据区。在文件中不占空间，但在加载到内存时会被分配并清零
<code>.init_array</code>	存放函数指针，这些函数会在库被加载（ <code>dlopen</code> ）时自动执行。分析反调试的绝佳入口点
<code>.fini_array</code>	存放函数指针，这些函数会在库被卸载（ <code>dlclose</code> ）时自动执行
<code>.dynsym</code>	动态符号表，包含了库中导出和导入的函数和变量名
<code>.dynstr</code>	字符串表， <code>.dynsym</code> 中的符号名称就存储在这里

加载与链接

System.loadLibrary()

在 Java/Kotlin 代码中，开发者通过 `System.loadLibrary("mylib")` 来加载一个名为 `libmylib.so` 的原生库。系统会在 `lib/` 目录下的相应 ABI 文件夹（如 `arm64-v8a`）中查找并加载该库。

JNI (Java Native Interface)

JNI 是连接 Java 世界和 Native (C/C++) 世界的桥梁，是 Android 逆向分析中的核心知识点。

JNI 基础概念

Java 侧声明：

```
public class NativeHelper {  
    static {  
        System.loadLibrary("native-lib"); // 加载 libnative-lib.so  
    }  
  
    // 静态 native 方法  
    public static native String doEncrypt(String input);  
  
    // 实例 native 方法  
    public native byte[] processData(byte[] data, int flag);  
  
    // 多参数 native 方法  
    public native int complexOperation(String str, int[] array, boolean flag);  
}
```

Native 侧实现：

```
// 静态方法 JNI 函数签名: 第二个参数是 jclass
JNIEXPORT jstring JNICALL Java_com_example_app_NativeHelper_doEncrypt(JNIEnv *env, jclass clazz, jstring input)
{
    const char *nativeString = (*env)->GetStringUTFChars(env, input, 0);
    // 执行加密逻辑...
    jstring result = (*env)->NewStringUTF(env, encrypted_result);
    (*env)->ReleaseStringUTFChars(env, input, nativeString);
    return result;
}

// 实例方法 JNI 函数签名: 第二个参数是 jobject
JNIEXPORT jbyteArray JNICALL Java_com_example_app_NativeHelper_processData(JNIEnv *env, jobject thiz, jbyteArray data, jint flag) {
    jsize len = (*env)->GetArrayLength(env, data);
    jbyte *body = (*env)->GetByteArrayElements(env, data, 0);

    // 处理数据...

    jbyteArray result = (*env)->NewByteArray(env, len);
    (*env)->SetByteArrayRegion(env, result, 0, len, processed_data);
    (*env)->ReleaseByteArrayElements(env, data, body, 0);
    return result;
}
```

JNI 数据类型映射

Java 类型	JNI 类型	签名字符
boolean	jboolean	Z
byte	jbyte	B
char	jchar	C
short	jshort	S
int	jint	I
long	jlong	J
float	jfloat	F
double	jdouble	D
void	void	V
String	jstring	Ljava/lang/String;
Object	jobject	L完整类名;
int[]	jintArray	[I
byte[]	jbyteArray	[B

JNI 常用函数

分类	函数
字符串操作	NewStringUTF(), GetStringUTFChars(), ReleaseStringUTFChars()
数组操作	NewByteArray(), GetByteArrayElements(), SetByteArrayRegion()
对象操作	NewObject(), GetObjectClass(), CallObjectMethod()
字段访问	GetFieldID(), GetIntField(), SetIntField()
方法调用	GetMethodID(), CallVoidMethod(), CallIntMethod()

JNI 方法注册

静态注册（编译时确定）：

函数名必须严格遵循命名规则： Java_包名_类名_方法名

```
JNICALL jstring Java_com_example_app_MainActivity_stringFromJNI(JNIEnv *env, jobject thiz) {
    return (*env)->NewStringUTF(env, "Hello from JNI!");
}
```

动态注册（运行时注册）：

```
// 定义方法映射表
static JNINativeMethod gMethods[] = {
    {"encrypt", "(Ljava/lang/String;)Ljava/lang/String;", (void*)native_encrypt},
    {"decrypt", "([B)[B", (void*)native_decrypt},
    {"init", "(I)V", (void*)native_init}
};

// JNI_OnLoad 函数在库加载时自动调用
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env;
    if ((*vm)->GetEnv(vm, (void**)&env, JNI_VERSION_1_6) != JNI_OK) {
        return JNI_ERR;
    }

    jclass clazz = (*env)->FindClass(env, "com/example/app/NativeHelper");
    if (clazz == NULL) {
        return JNI_ERR;
    }

    // 注册 native 方法
    if ((*env)->RegisterNatives(env, clazz, gMethods,
        sizeof(gMethods)/sizeof(gMethods[0])) < 0) {
        return JNI_ERR;
    }

    return JNI_VERSION_1_6;
}
```

动态链接器

动态链接器 (`/system/bin/linker` 或 `linker64`) 负责:

1. 将 `.so` 文件映射到进程的虚拟地址空间
2. 重定位符号 (调整代码中的地址引用)
3. 解析该库的依赖项 (即它需要从其他库, 如 `libc.so`, 导入的函数), 并填充导入函数的地址
4. 执行 `.init_array` 中的初始化函数

静态分析

识别关键函数

目标	方法
JNI 函数	在符号列表中搜索 <code>Java_</code> 前缀，可以快速定位所有 Java 和 Native 的交互点
导出函数	查看 <code>Exports</code> 列表，寻找有意义的函数名，如 <code>encrypt</code> , <code>decrypt</code> , <code>checkSignature</code>
<code>.init_array</code> 函数	查看 <code>.init_array</code> section，分析在库加载时自动执行的函数

命令行工具：

```
# 列出导出的 JNI 函数  
nm -D libexample.so | grep Java_  
  
# 使用 objdump  
objdump -T libexample.so | grep Java_
```

使用 IDA Pro / Ghidra

1. 加载文件：将 `.so` 文件拖入 IDA 或 Ghidra
2. 查看伪代码：按 `F5` (IDA) 或等待 Ghidra 的自动分析，直接阅读反编译出的 C 伪代码
3. 交叉引用 (Cross-References)：对一个函数名或字符串常量按 `X` 键，可以查看所有引用了它的地方
4. 图形模式：使用图形视图来理解复杂的函数调用流程和条件分支

动态分析

Frida Hook 原生函数

当静态分析困难（如代码被混淆或算法复杂）时，动态 Hook 是最有效的方法。

基于偏移地址 Hook：

```
// Frida 脚本
const baseAddr = Module.findBaseAddress("libnative-lib.so");
const targetFuncPtr = baseAddr.add(0x1234); // 0x1234 是从 IDA/Ghidra 获取的函数偏移

Interceptor.attach(targetFuncPtr, {
    onEnter: function(args) {
        // args[0], args[1]... 是函数参数 (指针)
        console.log("Hooked function called!");
        // 可以使用 Memory.readCString(args[0]) 等来读取参数内容
    },
    onLeave: function(retval) {
        // retval 是函数返回值
        // retval.replace(0x1); // 可以修改返回值
    }
});
});
```

基于导出名称 Hook：

```
var encrypt_func = Module.findExportByName("libnative.so",
    "Java_com_example_app_NativeHelper_doEncrypt");

Interceptor.attach(encrypt_func, {
    onEnter: function(args) {
        // args[0] = JNIEnv*
        // args[1] = jclass/jobject
        // args[2] = 第一个参数 (jstring)
        var jstring_ptr = args[2];
        var str_content = Java.vm.getEnv().getStringUtfChars(jstring_ptr, null);
        console.log("Input: " + str_content.readCString());
    },
    onLeave: function(retval) {
        var result = Java.vm.getEnv().getStringUtfChars(retval, null);
        console.log("Output: " + result.readCString());
    }
});
});
```

Java 层 Hook:

```
Java.perform(function() {
    var NativeHelper = Java.use("com.example.app.NativeHelper");

    NativeHelper.doEncrypt.implementation = function(input) {
        console.log("Java -> Native: " + input);
        var result = this.doEncrypt(input);
        console.log("Native -> Java: " + result);
        return result;
    };
});
```

Unidbg 模拟执行

Unidbg 是一个基于 Unicorn 的 Android 原生库模拟执行框架，可以在 PC 上模拟执行 `.so` 文件。

```
// Unidbg 基本用法
public class EmulatorDemo {
    public static void main(String[] args) {
        AndroidEmulator emulator = AndroidEmulatorBuilder.for64Bit().build();
        Memory memory = emulator.getMemory();
        memory.setLibraryResolver(new AndroidResolver(23));

        // 加载目标 so
        Module module = emulator.loadLibrary(new File("libtarget.so"));

        // 调用函数
        Number result = module.callFunction(emulator, "target_function", arg1, arg2);
        System.out.println("Result: " + result);
    }
}
```

常见保护手段

字符串加密

保护机制: `.so` 文件中的敏感字符串 (如密钥、URL) 被加密存放, 在运行时动态解密使用。静态分析时无法直接看到。

Native 实现示例:

```
// 加密的字符串
static char encrypted[] = "\x56\x2a\x3b\x4c\x5d"; // XOR 加密后的数据

const char* get_decrypted_string() {
    static char decrypted[256];
    int key = 0x42;
    for (int i = 0; encrypted[i]; i++) {
        decrypted[i] = encrypted[i] ^ key;
    }
    return decrypted;
}
```

攻击方法:

```
// Frida Hook 解密函数
var decrypt_func = Module.findExportByName("libnative.so", "decrypt_string");
Interceptor.attach(decrypt_func, {
    onLeave: function(retval) {
        console.log("Decrypted string:", Memory.readCString(retval));
    }
});
```

代码混淆

常见混淆技术:

- 控制流平坦化 (Control Flow Flattening)
- 虚假控制流 (Bogus Control Flow)
- 指令替换 (Instruction Substitution)
- 符号剥离 (Symbol Stripping)

攻击方法:

```
// 指令级 Hook 示例
var baseAddr = Module.findBaseAddress("libnative.so");
Interceptor.attach(baseAddr.add(0x1000), {
    onEnter: function(args) {
        console.log("Register state:", this.context);
    }
});
```

反调试技术

常见检测方法:

```
// ptrace 反调试
JNIEXPORT void JNICALL
Java_com_example_app_Security_checkDebugger(JNIEnv *env, jclass clazz) {
    if (ptrace(PTRACE_TRACE_ME, 0, 1, 0) == -1) {
        // 检测到调试器, 执行反制措施
        exit(1);
    }
}

// 模拟器检测
JNIEXPORT jboolean JNICALL
Java_com_example_app_Security_isEmulator(JNIEnv *env, jclass clazz) {
    char prop_value[256];
    __system_property_get("ro.kernel.qemu", prop_value);
    return strcmp(prop_value, "1") == 0;
}
```

绕过方法:

```
// 绕过反调试示例
var anti_debug = Module.findExportByName("libnative.so", "check_debug");
Interceptor.attach(anti_debug, {
    onLeave: function(retval) {
        retval.replace(0); // 返回 0 表示未检测到调试
    }
});

// Hook ptrace 系统调用
var ptrace = Module.findExportByName("libc.so", "ptrace");
Interceptor.attach(ptrace, {
    onEnter: function(args) {
        args[0] = ptr(0); // 修改 ptrace 参数
    }
});
```

init_array 详解

调用时机

.init_array 中的函数在 ELF 库加载过程中的早期阶段 被调用，这个时机非常关键，发生在 JNI_OnLoad 之前。

完整的调用流程

```
System.loadLibrary("native")
    ↓
nativeLoad() [art/runtime/native/java_lang_Runtime.cc]
    ↓
android_dlopen_ext() [bionic/libdl/libdl.cpp]
    ↓
do_dlopen() [bionic/linker/linker.cpp]
    ↓
find_library() → load_library() → link_image()
    ↓
call_constructors() → init_array 函数执行
    ↓
JNI_OnLoad() 执行
```

Linker 源码分析

```
// bionic/linker/linker.cpp
void soinfo::call_constructors() {
    // 1. 首先调用 DT_INIT 初始化函数
    if (init_func_ != nullptr) {
        init_func_();
    }

    // 2. 然后遍历 .init_array section 函数指针
    if (init_array_ != nullptr) {
        for (size_t i = 0; i < init_array_count_; ++i) {
            // 调用每个构造函数, 包括 init_string_obfuscation
            ((void (*)())init_array_[i])();
        }
    }
}
```

C/C++ 声明方式

```
// 字符串混淆初始化函数声明
__attribute__((constructor))
void init_string_obfuscation() {
    // 字符串解密和反调试逻辑
    decrypt_critical_strings();
    setup_anti_debug_measures();
}

// 可以指定优先级 (数字越小, 优先级越高)
__attribute__((constructor(101)))
void init_anti_debug_level1() {
    // 第一级反调试检测
    basic_environment_check();
}

__attribute__((constructor(102)))
void init_string_decryption() {
    // 字符串解密, 依赖第一级检测通过
    if (environment_safe) {
        decrypt_strings();
    }
}
```

实际应用示例

```
// 实际字符串混淆初始化函数示例
__attribute__((constructor(100)))
void init_string_obfuscation() {
    // 1. 环境安全检查
    if (detect_debug_environment()) {
        // 检测到调试环境，执行反制措施
        execute_anti_debug_response();
        return;
    }

    // 2. 解密关键字符串
    decrypt_api_strings();
    decrypt_config_strings();
    decrypt_url_strings();

    // 3. 标记初始化完成
    string_obfuscation_initialized = true;
}

// 字符串解密函数
void decrypt_api_strings() {
    for (int i = 0; i < API_STRING_COUNT; i++) {
        decrypt_string_xor(encrypted_api_names[i],
                           decrypted_api_names[i],
                           API_XOR_KEY);
    }
}

// XOR 解密实现
void decrypt_string_xor(const char* encrypted, char* decrypted, uint8_t key) {
    int len = strlen(encrypted);
    for (int i = 0; i < len; i++) {
        decrypted[i] = encrypted[i] ^ key;
    }
    decrypted[len] = '\0';
}
```

逆向分析对策

静态分析方法

```
# Python 脚本分析 .init_array section
from elftools.elf.elffile import ELFFile

def analyze_init_array(so_path):
    with open(so_path, 'rb') as f:
        elf = ELFFile(f)

        # 查找 .init_array section
        init_array_section = elf.get_section_by_name('.init_array')
        if init_array_section:
            data = init_array_section.data()

            print(f"[+] Found .init_array section, size: {len(data)} bytes")

            # 解析函数指针 (8 字节对齐, 64 位系统)
            for i in range(0, len(data), 8):
                if i + 8 <= len(data):
                    func_addr = int.from_bytes(data[i:i+8], 'little')
                    print(f"[+] Init function {i//8}: 0x{func_addr:x}")

    if __name__ == "__main__":
        analyze_init_array("libtarget.so")
```

动态分析方法

```
// Frida 脚本 Hook init_array 执行
function hookInitArray() {
    // Hook 构造函数调用函数
    var call_constructors = Module.findExportByName(
        "linker64",
        "_ZN6soinfo17call_constructorsEv"
    );

    if (call_constructors) {
        Interceptor.attach(call_constructors, {
            onEnter: function(args) {
                var soinfo = args[0];
                console.log("[+] Calling constructors for SO");
                this.start_time = Date.now();
            },
            onLeave: function(retval) {
                var duration = Date.now() - this.start_time;
                console.log("[+] Constructors completed in " + duration + "ms");
            }
        });
    }
}

// 直接 hook 目标SO 的 init_array 函数
hook_target_init_functions();
}

function hook_target_init_functions() {
    var target_module = Process.findModuleByName("libtarget.so");
    if (target_module) {
        // 根据静态分析结果 hook 特定地址的函数
        var init_func_addr = target_module.base.add(0x2000); // 示例地址

        Interceptor.attach(init_func_addr, {
            onEnter: function(args) {
                console.log("![!] init_string_obfuscation called");
                console.log("[+] Call stack:");
                console.log(Thread.backtrace(this.context, Backtracer.ACCURATE)
                    .map(DebugSymbol.fromAddress).join("\n"));
            },
            onLeave: function(retval) {
                console.log("![!] init_string_obfuscation completed");
            }
        });
    }
}

// 执行 hook
hookInitArray();
```

分析要点总结

特性	说明
执行时机早	在 <code>JNI_OnLoad</code> 之前执行，难以常规方式 Hook
优先级控制	可以通过参数控制多个初始化函数的执行顺序
静态分析困难	加密字符串在静态分析时不可见
调试时机窗口短	执行时间短，难以及时介入

总结

分析 Android `.so` 文件需要掌握以下核心技能：

1. ELF 格式理解：熟悉 ELF 文件结构，特别是关键 section
2. JNI 机制：理解 Java 和 Native 层的交互方式
3. 静态分析：使用 IDA Pro / Ghidra 进行反编译分析
4. 动态分析：使用 Frida 进行运行时 Hook 和调试
5. 保护绕过：了解常见保护手段并掌握绕过方法

通过综合运用这些技术，可以有效地分析和理解 Android 应用的原生代码逻辑。

[F08] ART 运行时

F08: Android 运行时 (ART) 深度解析

ART (Android Runtime) 是 Android 5.0 (Lollipop) 之后默认的应用程序运行时环境，取代了旧的 Dalvik 虚拟机 (DVM)。ART 的引入显著改变了 Android 应用的执行方式，旨在提高应用的性能、启动速度和电池续航。

目录

- android 运行时 (ART) 深度解析
 - ART (Android Runtime) 是 Android 5.0 (Lollipop) 之后默认的应用程序运行时环境，取代了旧的 Dalvik 虚拟机 (DVM)。ART 的引入显著改变了 Android 应用的执行方式，旨在提高应用的性能、启动速度和电池续航。
 - 目录
 - 核心机制: AOT vs JIT
 - Dalvik 的 JIT (Just-In-Time)
 - ART 的 AOT (Ahead-Of-Time)
 - 混合编译 (AOT + JIT)
 - ART 生成的文件格式
 - OAT 文件 (`.oat`)
 - VDEX 文件 (`.vdex`)
 - ART 文件 (`.art`) (Image)
 - ART vs. Dalvik
 - 对逆向工程的影响

核心机制：AOT vs JIT

!!! question "思考：为什么逆向工程师必须理解 ART？" 你可能会想："我只关心应用的 Java 代码和加密算法，ART 的编译机制与我有什么关系？"

实际场景告诉你答案：

- Frida Hook 失败：你写的 Hook 脚本在 Android 4.x 上好用，在 8.0+ 上就不工作了——因为 ART 的 AOT 编译改变了方法的执行方式
- 脱壳困境：你用传统方法 dump DEX，结果发现关键类根本不在 DEX 里——它们在运行时被解密后直接编译成了 OAT
- 性能分析：为什么同样的代码在不同 Android 版本上性能差异巨大？混合编译模式是关键
- 反调试对抗：某些 App 会检测 OAT 文件的完整性，或者利用 `dex2oat` 的时机来进行反调试

核心要点：

- Android 5.0+ 的应用不再是简单的"DEX 字节码"执行
- 真正执行的是 本地机器码（OAT 文件）
- 理解 DEX → VDEX → OAT 的转换流程，才能应对现代 Android 逆向

Dalvik 的 JIT (Just-In-Time)

在 Android 4.4 及更早版本中，Dalvik 虚拟机使用 JIT 编译。

- 工作方式：应用每次运行时，Dalvik 会解释执行 DEX 字节码。对于频繁执行的"热点代码"（hotspot），JIT 编译器会将其动态地编译成本地机器码并缓存。
- 优点：安装速度快，不占用额外存储空间。
- 缺点：应用启动和运行期间需要持续进行解释和编译，导致启动慢、耗电多。

ART 的 AOT (Ahead-Of-Time)

ART 最初的设计是纯 AOT 编译。

- 工作方式: 在应用安装时, 系统会调用 `dex2oat` 工具, 将 APK 中的 `classes.dex` 文件完整地编译成本地机器码, 并以 OAT 文件的形式存储。
- 优点:
 - 运行速度快: 应用直接执行本地机器码, 无需实时编译, 性能和启动速度都大大提升。
 - 更省电: CPU 在运行时负担更轻。
- 缺点:
 - 安装时间长: 应用安装过程需要额外的编译时间。
 - 占用空间大: 预编译的 OAT 文件会占用更多的存储空间。

混合编译 (AOT + JIT)

从 Android 7.0 (Nougat) 开始, ART 引入了结合 JIT 的混合编译模式, 以平衡上述优缺点。

- 工作流程:
 1. 初次安装: 应用安装速度很快, 不进行 AOT 编译。
 2. 首次运行: 应用代码由解释器执行, 同时 JIT 编译器会介入, 编译热点代码。在此期间, ART 会生成一份代码执行频率的分析文件 (Profile)。
 3. 设备空闲时: 当设备处于空闲状态并正在充电时, Android 系统会启动一个后台优化任务。该任务会根据之前收集的 Profile 信息, 只对那些频繁执行的热点方法进行 AOT 编译, 并生成新的 OAT 文件。
 - 优点: 兼顾了安装速度、运行性能和存储占用, 是目前 Android 的标准执行模式。
-

ART 生成的文件格式

当 ART 处理一个 APK 时，会在 `/data/dalvik-cache/<arch>/` 目录下生成一些优化后的文件。

OAT 文件 (`.oat`)

OAT (Optimized Android file format) 文件是核心。它包含了由 `dex2oat` 从 DEX 字节码编译而来的本地机器码 (ARM 汇编)。一个 OAT 文件通常包含：

- oatdata: 包含已编译的本地代码。
- oatexec: 包含可执行的本地代码。
- oatlastword: 标识 OAT 文件的结束。

VDEX 文件 (`.vdex`)

从 Android 8.0 (Oreo) 开始引入。为了进一步优化，系统会将原始的 `classes.dex` 文件进行“解压缩”和“验证”，生成一个 VDEX (Verified DEX) 文件。这样做的好处是：

- 快速验证：系统可以直接使用 VDEX 文件，跳过了对原始 DEX 的验证步骤，加快了加载速度。
- 内容：VDEX 文件本质上是一个未压缩的、带有额外依赖和验证信息的 DEX 文件。
`dex2oat` 会直接使用 VDEX 文件作为输入来生成 OAT 文件。

ART 文件 (`.art`) (Image)

这是一个预加载的镜像文件，包含了系统启动时需要预加载的一些核心类（如 `java.lang.Object`）。当 Zygote 进程启动时，会直接将这个镜像映射到内存，从而避免了对这些常用类进行重复的初始化，加快了所有应用的启动速度。

总结：在现代 Android 系统中，执行流程是：`classes.dex` → (安装时) `.vdex` → (后台优化时) `.oat`。

ART vs. Dalvik

特性	ART	Dalvik
编译模式	AOT + JIT 混合编译	JIT
执行单元	本地机器码 (主要)	DEX 字节码
性能	更高	较低
启动速度	更快	较慢
安装时间	更快 (混合模式下)	快
存储占用	更高 (因 OAT 文件)	较低
垃圾回收 (GC)	优化更好, 暂停时间更短	效率较低

对逆向工程的影响

!!! tip "实战技巧: 从 ART 机制找到突破口" 理解 ART 的工作原理, 能让你找到很多"非常规"的逆向思路:

脱壳新思路:

1. 监控 `dex2oat` 调用: 某些壳会在运行时动态调用 `dex2oat`, 监控其命令行参数能发现隐藏的 DEX
2. 从 VDEX 提取 DEX: Android 8.0+ 的 VDEX 文件本质上就是 DEX, 用 `vdexExtractor` 可以快速提取
3. 从 OAT 还原 DEX: 使用 `oat2dex` 等工具从编译后的 OAT 文件反推原始 DEX

Hook 优化策略:

- Java 方法 Hook: 优先 Hook Java 层 API, 更稳定通用

-
- Native Hook: 当 Java Hook 失效时, 找到 ART 编译后的机器码地址进行 inline hook
 - GOT/PLT Hook: Hook 动态链接库的导入表, 绕过代码完整性检查
 - Hook 点的变化: 由于存在 AOT 编译, Frida/Xposed 等框架的 Hook 原理也需要适应。它们不仅仅是 Hook Java 方法, 实际上是找到了该方法编译后的本地机器码地址, 并对其进行修改 (inline hook)。
 - 脱壳的复杂性: 许多加固厂商利用 ART 的 AOT 机制。他们可能会在运行时动态解密并加载 DEX, 然后手动调用 `dex2oat` 生成 OAT 文件来执行。这使得传统的 DEX Dump 方法失效, 需要对 OAT 文件格式和 `dex2oat` 的调用时机进行监控。
 - OAT 文件分析: 高级逆向分析有时需要直接分析 OAT 文件。有一些工具 (如 `oatdump`) 可以从 OAT 文件中提取出原始的 DEX 数据或查看编译后的汇编代码。
 - 寻找代码的源头: 即使代码被 AOT 编译, 其元数据依然与原始的 DEX 文件相关联。因此, 我们的分析起点通常还是从 `classes.dex` 反编译出的 Java 代码开始, 而不是直接一头扎进 OAT 文件的汇编代码中。

[F09] ARM 汇编入门

F09: ARM 汇编入门 (Android Native)

当应用的核心逻辑、加密算法或性能密集型任务用 C/C++ 编写时，它们会被编译成原生库 (.so 文件)。在 Android 上，这些库主要是 ARM 架构的。理解 ARM 汇编是分析 .so 文件的基础。本指南将介绍逆向工程师需要了解的 ARMv7 (32-bit) 和 ARMv8 (64-bit/AArch64) 的基础知识。

!!! question "思考：为什么必须学习汇编？" 当你遇到以下场景时，该如何应对？

- 用 JADX 打开 APK，发现关键的加密逻辑都在 native 方法中
- Frida Hook 到了某个 JNI 函数，但参数是指针，不知道如何读取
- IDA 打开 .so 文件，满屏的汇编指令让你无从下手

这些场景的共同点是：核心逻辑被编译成了机器码。不理解汇编，就像试图在不懂外语的情况下阅读外文书籍——你只能靠猜。

目录

1. 基本概念：ARM vs x86

2. 寄存器 (Registers)

- ARM 32-bit (ARMv7)

- ARM 64-bit (AArch64)

1. 核心指令集

- 数据移动指令

-
- 算术与逻辑指令
 - 分支与条件执行指令
 - 栈操作指令
1. 函数调用约定 (Procedure Call Standard)
 2. 从 C 代码到汇编
 3. IDA Pro/Ghidra 中的视图
-

基本概念：ARM vs x86

- RISC vs CISC: ARM 是精简指令集计算机 (RISC)，指令长度固定，种类较少，操作简单。
x86 是复杂指令集计算机 (CISC)。
 - Load/Store 架构: ARM 是一种"加载/存储"架构。这意味着数据处理（如加法、减法）只能在寄存器之间进行。你必须先用加载指令（LDR）将内存中的数据加载到寄存器，计算完成后用存储指令（STR）将结果存回内存。
 - 指令模式: ARMv7 (32-bit) 支持两种指令集:
 - ARM: 32-bit 定长指令，功能强大。
 - Thumb: 16-bit/32-bit 变长指令，代码密度更高，是移动设备上的主流。在 IDA 等工具中，你通常会分析 Thumb 模式下的代码。
-

寄存器 (Registers)

寄存器是 CPU 内的高速存储单元。

!!! tip "快速定位关键寄存器" 在分析一个陌生函数时，如何快速抓住重点？

- 函数入口: 先看 R0-R3 (32 位) 或 X0-X7 (64 位)，这些是参数
 - 函数返回: 关注 R0/X0，这是返回值存放的地方
-

- 函数调用：`BL` 指令前后，检查参数寄存器的变化
- 栈操作：`SP` 的变化反映了局部变量的分配

这种“重点优先”的阅读策略，能让你快速理解函数的输入输出，而不必逐行分析每条指令。

ARM 32-bit (ARMv7)

共有 16 个通用寄存器 (`R0` - `R15`)。

寄存器	别名	用途
<code>R0</code> - <code>R3</code>	<code>A1</code> - <code>A4</code>	参数寄存器 (Argument)。用于传递函数的前 4 个参数。 <code>R0</code> 也用作返回值寄存器。
<code>R4</code> - <code>R11</code>	<code>V1</code> - <code>V8</code>	变量寄存器 (Variable)。用于保存函数的局部变量。
<code>R12</code>	<code>IP</code>	过程调用间临时寄存器 (Intra-Procedure call scratch register)。
<code>R13</code>	<code>SP</code>	栈指针 (Stack Pointer)。指向栈顶。
<code>R14</code>	<code>LR</code>	链接寄存器 (Link Register)。存储函数调用的返回地址。
<code>R15</code>	<code>PC</code>	程序计数器 (Program Counter)。指向当前正在执行的指令。

ARM 64-bit (AArch64)

寄存器数量更多，且功能更明确。

| 寄存器 | 用途 | | :----- | :----- |
----- | | `X0` - `X7` | 参数寄存器。用于传递函数的前 8 个参数。`X0` 同样用作返回值寄存器。 | | `X8` - `X18` | 调用者/被调用者保存的临时寄存器。 | | `X19` - `X28` | 被调用者保存的寄存器 (Callee-saved)。 | | `X29` | `FP` | 帧指针 (Frame Pointer)。 | | `X30` | `LR` | 链接寄存器 (Link Register)。 | | `SP` | 栈指针 (Stack Pointer)。 |

- 注*: `W` 寄存器 (`W0`, `W1`...) 是 `X` 寄存器的低 32 位。例如，对 `W0` 的操作就是对 `X0` 的低 32 位进行操作。

核心指令集

数据移动指令

- `MOV R1, R2` (32-bit) / `MOV X1, X2` (64-bit): 将寄存器 `R2` 的值移动到 `R1`。
- `LDR R0, [SP, #4]` (32-bit) / `LDR X0, [SP, #8]` (64-bit): 加载。从栈指针 `SP` 偏移 4 (或 8) 字节的位置读取数据，并存入 `R0` (或 `X0`)。
- `STR R0, [SP, #4]` (32-bit) / `STR X0, [SP, #8]` (64-bit): 存储。将 `R0` (或 `X0`) 的值写入到 `SP` 偏移 4 (或 8) 字节的内存地址。
- `ADR X0, aHelloWorld` (64-bit, PC-relative): `ADR` (Address PC-Relative) 指令将一个相对于 PC 的地址 (如字符串 "Hello World" 的地址) 加载到 `X0`。

算术与逻辑指令

- `ADD R0, R1, R2` : $R0 = R1 + R2$ 。
- `SUB R0, R1, #1` : $R0 = R1 - 1$ 。
- `and R0, R0, #0xFF` : 按位与。
- `CMP R0, #10` : 比较 `R0` 和 10。该指令会更新状态寄存器 (CPSR)，但不存储结果。它总是紧跟在条件分支指令之前。

分支与条件执行指令

- `B label` : 分支 (Branch)。无条件跳转到 `label`。
- `BL label` : 带链接的分支 (Branch with Link)。跳转到 `label` 之前，将下一条指令的地址存入 `LR` (链接寄存器)。这是函数调用的核心指令。
- `BX LR` / `RET` : 带交换的分支 (Branch with Exchange) / 返回。跳转到 `LR` 中的地址，实现函数返回。`RET` 是 `BX LR` 的别名。
- `B.EQ label` : 条件分支。如果前一个 `CMP` 指令的结果是相等 (Equal)，则跳转。

- `B.NE label`: 不相等 (Not Equal)。
- `B.GT label`: 大于 (Greater Than)。
- `B.LT label`: 小于 (Less Than)。
- `B.GE label`: 大于或等于 (Greater or Equal)。
- `B.LE label`: 小于或等于 (Less or Equal)。

栈操作指令

- `PUSH {R4, LR}`: 将 `R4` 和 `LR` 寄存器压入栈。通常在函数开头，用于保存需要使用的寄存器和返回地址。
- `POP {R4, PC}`: 将栈顶数据弹出到 `R4` 和 `PC`。`POP {..., PC}` 是一种常见的函数返回方式，它将保存在栈上的 `LR` 值直接弹出到 `PC`，实现了跳转返回。
- `STP X29, X30, [SP, #-16]!` (A64): `STP` (Store Pair) 指令，将一对寄存器 (`X29 / FP`, `X30 / LR`) 存入 `SP` 指向的地址，`!` 表示 `SP` 会预先减去 16。
- `LDP X29, X30, [SP], #16` (A64): `LDP` (Load Pair) 指令，从 `SP` 地址加载数据到 `X29` 和 `X30`，然后 `SP` 再增加 16。

函数调用约定 (Procedure Call Standard)

1. 参数传递:

- 32-bit: 前 4 个参数通过 `R0` - `R3` 传递。
- 64-bit: 前 8 个参数通过 `X0` - `X7` 传递。
- 超出数量的参数通过栈传递。

1. 函数调用: 调用者使用 `BL` 指令。

2. 函数序言 (Prologue):

- 被调用函数 (子函数) 首先要做的是保存现场。

-
- 使用 `PUSH` 或 `STP` 将需要在函数中使用的寄存器（如 `R4-R11`, `FP`, `LR`）压入栈中。
 - 分配栈空间给局部变量 (`SUB SP, SP, #...`)。

1. 函数结语 (Epilogue):

- 函数执行完毕，准备返回。
- 释放局部变量的栈空间 (`ADD SP, SP, #...`)。
- 使用 `POP` 或 `LDP` 从栈中恢复之前保存的寄存器。
- 使用 `BX LR` 或 `RET` 或 `POP {PC}` 返回。

1. 返回值:

- 简单的返回值（整数、指针）存放在 `R0` (32-bit) 或 `X0` (64-bit) 中。
-

从 C 代码到汇编

C 代码:

```
int add_one(int a) {
    return a + 1;
}
```

`BX LR ; return`

```
RET ; return
```

- 伪代码视图 (Pseudocode View): IDA Pro (F5 键) 和 Ghidra 的反编译器可以直接将汇编代码转换成可读性很高的 C 伪代码，这是静态分析的利器。通常先看伪代码，遇到不理解的地方再回头看汇编。

[F10] x86 与 ARM 汇编基础

F10: x86 与 ARM 汇编基础指南

汇编语言是与计算机硬件直接对话的低级编程语言，是逆向工程、系统编程和性能优化的基石。在当今世界，x86 和 ARM 是两种最主流的指令集架构 (ISA)。理解它们的核心概念与差异对于逆向工程师至关重要。

- x86: 由 Intel 主导，采用CISC (复杂指令集计算机) 设计。指令长度可变，功能强大但复杂，主要用于桌面和服务器。
- ARM: 由 ARM Holdings 设计，采用RISC (精简指令集计算机) 设计。指令长度固定，设计简洁优雅，功耗低，主宰了移动和嵌入式设备领域。

目录

- [x86 与 ARM 汇编基础指南](#)
 - [目录](#)
 - [x86 汇编 \(IA-32\)](#)
 - [核心寄存器](#)
 - [常用指令](#)
 - [调用约定 \(Calling Convention\)](#)
 - [ARM 汇编 \(ARMv7\)](#)
 - [核心寄存器](#)
 - [加载/存储 \(Load/Store\) 架构](#)
 - [常用指令](#)
 - [调用约定 \(AAPCS\)](#)
 - [x86 vs. ARM 核心差异对比](#)

x86 汇编 (IA-32)

以 32 位 x86 架构为例，其设计复杂而灵活。

核心寄存器

8 个 32 位通用寄存器，它们有主要用途，但在很多情况下可以通用。

寄存器	主要用途
EAX	累加器 (Accumulator): 通常用于存放函数返回值和算术运算结果。
EBX	基址 (Base): 常作为数据段的基址指针。
ECX	计数器 (Counter): 常用于循环计数。
EDX	数据 (Data): 常用于存放数据，特别是在乘除法中与 EAX 配合。
ESP	栈指针 (Stack Pointer): 永远指向栈顶。
EBP	基址指针 (Base Pointer): 永远指向当前函数栈帧的底部。
ESI	源变址 (Source Index): 字符串和内存操作中的源地址。
EDI	目的变址 (Destination Index): 字符串和内存操作中的目的地址。
EIP	指令指针 (Instruction Pointer): 永远指向下一条将要执行的指令的地址。

常用指令

- 数据传送:
 - `MOV dest, src`: 将 `src` 的值赋给 `dest`。 (e.g., `MOV EAX, EBX`)
 - `PUSH val`: 将 `val` 压入栈顶，`ESP` 减 4。

- `POP reg`: 从栈顶弹出一个值到 `reg`, `ESP` 加 4。
- `LEA reg, [mem]`: 将 `mem` 的有效地址加载到 `reg`, 而不是其内容。
- 算术运算:
 - `ADD dest, src`: `dest = dest + src`
 - `SUB dest, src`: `dest = dest - src`
 - `INC reg`: `reg = reg + 1`
 - `DEC reg`: `reg = reg - 1`
- 逻辑与跳转:
 - `CMP reg1, reg2`: 比较 `reg1` 和 `reg2` (实际是做减法), 并根据结果设置标志位。
 - `JMP target`: 无条件跳转到 `target` 地址。
 - `JE target`: 如果相等 (Zero Flag=1) 则跳转。
 - `JNE target`: 如果不相等 (Zero Flag=0) 则跳转。
 - `JG/JL/JGE/JLE`: 大于/小于/大于等于/小于等于时跳转。
- 函数调用:
 - `CALL target`: 将 `EIP` 的下一条指令地址压栈, 然后跳转到 `target`。
 - `RET`: 从栈顶弹出地址, 并跳转到该地址。

调用约定 (Calling Convention)

规定了函数如何传递参数和返回结果。常见于 32 位 Windows 的是 `stdcall`, 而 Linux/macOS 上常见 `cdecl`。

- `cdecl`:
 - 参数从右到左依次压入栈中。

- 调用者负责在函数返回后清理栈。
- **stdcall**:
- 参数从右到左依次压入栈中。
- 被调用者自己负责在返回前清理栈。

ARM 汇编 (ARMv7)

以 32 位 ARM 架构为例，其设计简洁而高效。

核心寄存器

共有 16 个 32 位通用寄存器 (R0-R15)。

寄存器	别名	主要用途
R0 - R3		参数/返回值: 用于传递函数的前 4 个参数, R0 也用于存放函数返回值。
R4 - R12		通用寄存器, 用于保存局部变量。
R13	SP	栈指针 (Stack Pointer): 指向栈顶。
R14	LR	链接寄存器 (Link Register): 存储函数的返回地址。
R15	PC	程序计数器 (Program Counter): 指向下一条将要执行的指令。

加载/存储 (Load/Store) 架构

这是 RISC 的核心思想。CPU 不能直接对内存中的数据进行运算。

1. 必须先用 **LDR** (Load Register) 指令将内存中的数据加载到寄存器中。
2. 在寄存器之间完成所有算术和逻辑运算。

3. 再用 `STR` (Store Register) 指令将结果存回内存。

常用指令

· 数据传送:

· `MOV Rd, Rn` : 将 `Rn` 的值赋给 `Rd`。 (e.g., `MOV R0, R1`)

· 算术运算:

· `ADD Rd, Rn, Rm` : $Rd = Rn + Rm$

· `SUB Rd, Rn, Rm` : $Rd = Rn - Rm$

· 内存操作:

· `LDR Rd, [Rn, #offset]` : 从地址 `Rn + offset` 加载一个字到 `Rd`。

· `STR Rd, [Rn, #offset]` : 将 `Rd` 的值存储到一个字到地址 `Rn + offset`。

· 栈操作:

· `PUSH {reg_list}` : 将寄存器列表压入栈。

· `POP {reg_list}` : 将值从栈中弹出到寄存器列表。

· 跳转与比较:

· `CMP Rn, Rm` : 比较 `Rn` 和 `Rm`，并设置标志位。

· `B target` : 无条件跳转到 `target`。

· `BEQ target` : 如果相等则跳转。

· `BNE target` : 如果不相等则跳转。

· `BL target` : (Branch with Link) "调用函数"。它会自动将下一条指令的地址存入 `LR` 寄存器，然后跳转到 `target`。

· 函数返回时，只需执行 `MOV PC, LR` 或 `BX LR` 即可。

调用约定 (AAPCS)

ARM Procedure Call Standard。

- 参数传递:
 - 前 4 个参数通过 R0, R1, R2, R3 传递。
 - 剩余的参数通过栈传递。
 - 返回值:
 - 返回值存储在 R0 中。
 - 返回地址:
 - 通过 LR 寄存器管理。
-

x86 vs. ARM 核心差异对比

特性	x86 (CISC)	ARM (RISC)
指令集	复杂, 长度可变	精简, 长度固定
内存访问	可以直接对内存操作 (e.g., ADD [mem], EAX)	加载/存储架构 (必须先 LDR, 再 STR)
寄存器	较少, 且有特定用途	较多, 大多为通用寄存器
函数调用	CALL 指令压栈 EIP	BL 指令将返回地址存入 LR 寄存器
参数传递	主要通过栈	主要通过寄存器 (R0-R3)
条件执行	通过 CMP 和 Jcc 跳转指令	所有指令都可以是条件执行的 (e.g., MOVEQ R0, R1)

TOTP 时间动态密码

TOTP 技术原理

TOTP (Time-Based One-Time Password) 是一种基于时间的一次性密码算法，广泛应用于双因素认证 (2FA) 和 API 防护场景。

核心原理

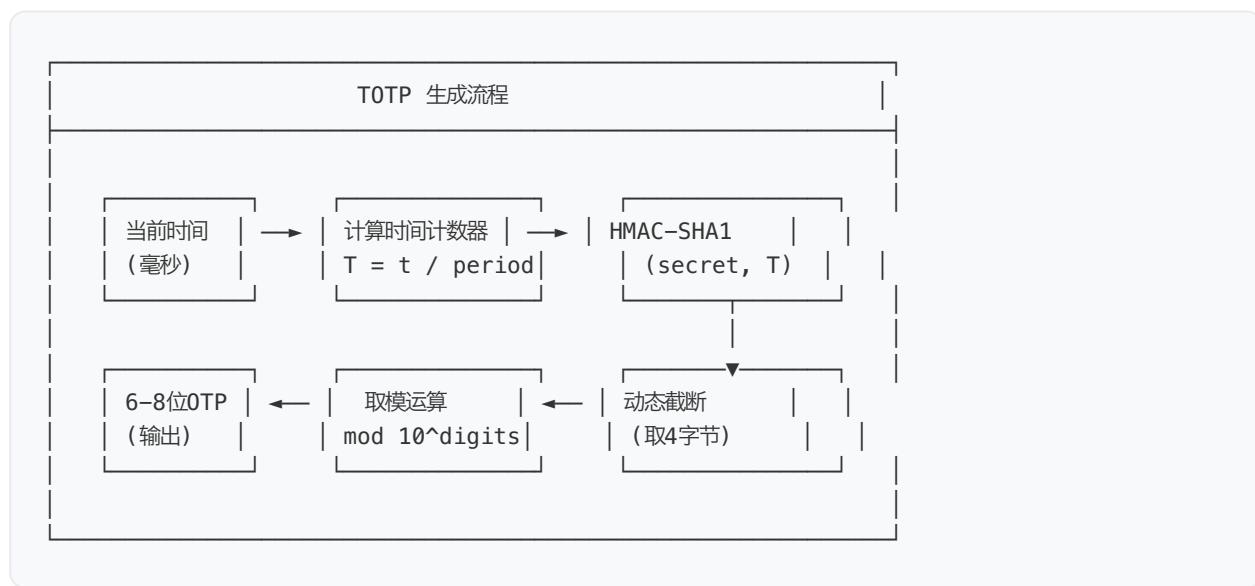
TOTP 基于 HMAC-SHA1 算法，结合共享密钥和当前时间戳生成动态密码。其核心公式为：

```
TOTP = HMAC-SHA1(Secret, TimeCounter) mod 10^Digits
```

其中：

- `TimeCounter = floor(.currentTimeMillis() / Period)`
- `Period` 通常为 30 秒
- `Digits` 通常为 6 位

算法流程



关键参数

参数	说明	典型值
Secret	共享密钥 (Base32编码)	16-32 字节
Period	时间步长	30 秒
Digits	OTP 位数	6-8 位
Algorithm	哈希算法	SHA1 / SHA256 / SHA512

标准实现

以下是符合 RFC 6238 规范的 Python 实现:

```
import hmac
import hashlib
import time
import base64

def generate_totp(secret: bytes, period: int = 30, digits: int = 6) -> str:
    """
    生成 TOTP 一次性密码

    Args:
        secret: 共享密钥 (原始字节)
        period: 时间步长 (秒)
        digits: OTP 位数

    Returns:
        生成的 OTP 字符串
    """
    # 1. 计算时间计数器
    counter = int(time.time()) // period
    counter_bytes = counter.to_bytes(8, byteorder='big')

    # 2. HMAC-SHA1 计算
    hmac_result = hmac.new(secret, counter_bytes, hashlib.sha1).digest()

    # 3. 动态截断 (Dynamic Truncation)
    offset = hmac_result[-1] & 0x0F
    binary = (
        (hmac_result[offset] & 0x7F) << 24 |
        (hmac_result[offset + 1] & 0xFF) << 16 |
        (hmac_result[offset + 2] & 0xFF) << 8 |
        (hmac_result[offset + 3] & 0xFF)
    )

    # 4. 取模得到最终 OTP
    otp = binary % (10 ** digits)
    return str(otp).zfill(digits)

def generate_totp_from_base32(secret_base32: str, period: int = 30, digits: int = 6) -> str:
    """
    从 Base32 编码的密钥生成 TOTP

    Args:
        secret_base32: Base32 编码的共享密钥
        period: 时间步长 (秒)
        digits: OTP 位数

    Returns:
        生成的 OTP 字符串
    """
    # 解码 Base32 密钥
```

```
secret = base64.b32decode(secret_base32.upper())
return generate_totp(secret, period, digits)

# 使用示例
if __name__ == "__main__":
    # Base32 编码的测试密钥
    test_secret = "JBSWY3DPEHPK3PXP"
    otp = generate_totp_from_base32(test_secret)
    print(f"Generated OTP: {otp}")
```

动态截断详解

动态截断 (Dynamic Truncation) 是 TOTP 的关键步骤，用于从 20 字节的 HMAC 结果中提取 4 字节作为 OTP 的来源：

1. 取偏移量：从 HMAC 结果的最后一个字节取低 4 位作为偏移量 (0-15)
2. 提取 4 字节：从偏移量位置开始取 4 个字节
3. 屏蔽符号位：将第一个字节的最高位置 0，确保结果为正数
4. 取模运算：对 10^{digi} 取模，得到最终的 OTP

```
# 动态截断伪代码
offset = hmac_result[-1] & 0x0F  # 取最后字节的低4位
binary = hmac_result[offset:offset+4]  # 取4字节
binary[0] &= 0x7F  # 屏蔽符号位
otp = int.from_bytes(binary, 'big') % (10 ** digits)
```

在反爬场景中的应用

1. 动态 Token 生成

服务端和客户端共享密钥，客户端生成的 TOTP 作为请求签名的一部分：

```
def sign_request(params: dict, secret: bytes) -> dict:  
    """使用 TOTP 签名请求"""  
    totp = generate_totp(secret)  
    params['timestamp'] = int(time.time()) * 1000  
    params['otp'] = totp  
    return params
```

2. 防重放攻击

由于 OTP 有时效性 (通常 30 秒), 捕获的请求无法长期复用。服务端验证时通常允许 ± 1 个周期的容差:

```
def verify_totp(secret: bytes, otp: str, tolerance: int = 1) -> bool:  
    """验证 TOTP, 允许时间误差"""  
    for offset in range(-tolerance, tolerance + 1):  
        counter = (int(time.time()) // 30) + offset  
        expected = generate_totp_with_counter(secret, counter)  
        if otp == expected:  
            return True  
    return False
```

3. 密钥保护

共享密钥通常经过混淆或加密存储在客户端, 增加逆向难度:

- 硬编码混淆: 密钥被拆分或异或后存储
- 加密配置: 密钥加密存储在配置文件中
- 远程获取: 启动时从服务端安全获取密钥

逆向要点

1. 定位密钥存储

```
// 常见的密钥存储位置
1. 硬编码在 JS/SO 中
2. 存储在 SharedPreferences / localStorage
3. 从服务端 API 动态获取
4. 使用设备信息派生 (PBKDF2, HKDF)
```

2. 确定算法参数

```
# 需要确定的参数
1. Period (时间步长): 常见值 30, 60
2. Digits (位数): 常见值 6, 8
3. Algorithm (算法): SHA1, SHA256, SHA512
4. 时间基准: Unix 时间戳 vs 自定义纪元
```

3. 时间同步

```
# 注意事项
1. 客户端和服务端时间可能存在偏差
2. 服务端通常允许 ±1 个周期的容差
3. 某些实现使用毫秒级时间戳
4. 注意时区问题
```

与 HOTP 的区别

特性	TOTP	HOTP
计数器	基于时间	基于事件 (递增)
有效期	时间窗口内	直到使用
同步问题	需要时间同步	需要计数器同步
适用场景	双因素认证	硬件令牌
RFC 规范	RFC 6238	RFC 4226

参考资料

- [RFC 6238 - TOTP: Time-Based One-Time Password Algorithm](#)
- [RFC 4226 - HOTP: An HMAC-Based One-Time Password Algorithm](#)
- [Google Authenticator 实现](#)

进阶主题

[A01] Android 沙箱实现

A01: Android 沙箱技术与实现指南

Android 沙箱技术，通常也被称为"虚拟化引擎"或"App 多开框架"，是一种在单个 Android 设备上创建隔离环境以运行其他应用程序的技术。它允许一个"宿主"应用程序在自己的进程空间内加载并运行一个"插件"应用程序，同时对插件应用的所有系统交互进行拦截和管理。

这项技术的核心应用包括：应用多开、无感知隐私保护、自动化测试、以及免安装运行 App。

目录

1. 核心概念：沙箱 vs. 虚拟机
2. 沙箱实现原理详解
 - android 沙箱技术与实现指南
 - 这项技术的核心应用包括：应用多开、无感知隐私保护、自动化测试、以及免安装运行 App。
 - 目录
 - 核心概念：沙箱 vs. 虚拟机
 - 沙箱实现原理详解
 - 1. 类加载 (Class Loading)
 - 2. 组件生命周期管理 (Component Lifecycle)
 - 3. 系统服务 Hook (API Hooking via Binder Proxy)
 - 4. 资源管理 (Resource Management)
 - 实现一个最小化沙箱的步骤

-
- 至此，插件 Activity 的界面就显示出来了，但它实际上是运行在 `StubActivity` 的"壳"里。

- 知名开源项目参考
- 挑战与局限

- 实现一个最小化沙箱的步骤
 - 知名开源项目参考
 - 挑战与局限
-

核心概念：沙箱 vs. 虚拟机

- 虚拟机 (VM): 创建一个完整的、独立的操作系统，拥有自己的内核和系统服务，资源开销巨大。
 - Android 沙箱: 不创建独立的操作系统。它运行在宿主 App 的进程中，与宿主共享同一个 Android 系统内核和运行时。它通过 API Hooking 的方式，为插件 App 创造了一个"虚拟的"运行环境，拦截和重定向其对系统服务的请求。本质上是一种进程内虚拟化。
-

沙箱实现原理详解

实现一个 Android 沙箱需要解决四大核心问题：

1. 类加载 (Class Loading)

由于插件 App 并未被系统"安装"，其代码不能通过常规的 `PathClassLoader` 加载。

- 解决方案：使用 `DexClassLoader`。宿主 App 需要创建一个 `DexClassLoader` 实例，将插件 APK 的路径和宿主 App 的私有数据目录（用于存放优化后的 ODEX 文件）作为参数传入。这样，宿主 App 就能加载并实例化插件 App 中的任意类。
-

2. 组件生命周期管理 (Component Lifecycle)

插件 App 的组件 (Activity, Service 等) 并没有在宿主 App 的 `AndroidManifest.xml` 中注册, 因此无法被系统直接启动。

- 解决方案: 占坑 (Stub/Proxy Component)。

1. 在宿主中预注册: 在宿主 App 的 `AndroidManifest.xml` 中预先注册一系列"占坑"的组件, 例如 `StubActivity1`, `StubActivity2`, `StubService1` ...
2. 请求拦截与替换: 当插件 App 想要启动一个组件时 (例如 `startActivity(intentToPluginActivity)`) , 这个请求会被我们下一步要讲的系统服务 Hook 拦截到。
3. 移花接木: 拦截到请求后, 沙箱框架会创建一个指向"占坑"Activity 的新 `Intent` (`intentToStubActivity`), 并将原始的 `Intent` 作为 extra 数据附加到新 `Intent` 上。然后, 它会用这个新的 `Intent` 去调用原始的系统服务。
4. 生命周期委托: 系统启动了 `StubActivity`。在 `StubActivity` 的 `onCreate` 方法中, 它会从 extra 中恢复出原始 `Intent`, 得知自己需要扮演哪个插件 Activity 的角色。然后, 它使用第一步的 `DexClassLoader` 实例化真正的插件 Activity, 并手动调用其 `onCreate`, `onStart`, `onResume` 等所有生命周期方法, 将自己的生命周期"委托"给插件 Activity。

3. 系统服务 Hook (API Hooking via Binder Proxy)

这是整个沙箱技术最核心、最复杂的部分。插件 App 的所有行为, 如启动 Activity、发送广播、访问数据库, 都是通过调用系统服务完成的。我们必须拦截这些调用。

- 目标: Android 的各种 `XXXManager` (如 `ActivityManager`, `PackageManager`) 实际上都是通过 Binder IPC 与系统服务 (`ActivityManagerService`, `PackageManagerService`) 通信的。我们需要 Hook 的就是这个通信的接口。
- 解决方案: 动态代理 (Dynamic Proxy)。

1. 定位 Binder 接口: 使用 Java 反射找到 `ActivityManager` 等类中持有的 `IActivityManager` 类型的 Binder 代理对象。
2. 创建代理对象: 使用 `java.lang.reflect.Proxy.newProxyInstance()` 方法, 为原始的 `IActivityManager` Binder 代理对象创建一个动态代理。

-
3. 实现 `InvocationHandler`: 在 `InvocationHandler` 的 `invoke` 方法中, 我们可以拦截所有对 `IActivityManager` 接口的方法调用 (如 `startActivity`, `getRunningAppProcesses` 等)。
 4. 请求重定向: 在 `invoke` 方法中, 判断当前请求是否来自插件 App。如果是, 就不执行原始的系统调用, 而是将其重定向到我们自己的沙箱管理逻辑中 (例如, 执行上述的"占坑"流程)。如果不是, 就调用原始的 Binder 方法, 保证宿主 App 自身功能正常。

4. 资源管理 (Resource Management)

插件 App 需要加载自己的布局、字符串、图片等资源。

- 解决方案: 创建一个自定义的 `Resources` 对象。
1. 通过 `AssetManager` 的隐藏方法 `addAssetPath()`, 将插件 APK 的路径添加到 `AssetManager` 中。
 2. 基于这个 `AssetManager` 创建一个新的 `Resources` 对象。
 3. 在创建插件 Activity 等组件时, 将这个自定义的 `Resources` 对象注入到其 `Context` 中, 从而让它可以访问到自己的资源。
-

实现一个最小化沙箱的步骤

以下是一个启动插件 Activity 的极简流程:

1. 准备:
 - 一个宿主 App。
 - 一个插件 App 的 APK 文件。
 - 在宿主 App 的 `AndroidManifest.xml` 中注册一个 `StubActivity`。
 1. Hook AMS: 在宿主 App 启动时 (如 `Application.onCreate`) , 通过反射和动态代理, Hook `IActivityManager`。
-

2. 加载插件: 当用户触发"启动插件"操作时:

- 创建 `DexClassLoader` 和自定义 `Resources` 对象。
- 构造一个指向插件主 Activity 的 `Intent`。
- 调用 `startActivity(pluginIntent)`。

1. 拦截与替换:

- `IActivityManager` 的动态代理 `invoke` 方法拦截到这个 `startActivity` 调用。
- `invoke` 方法发现这是一个插件 `Intent`，于是将其替换为一个指向 `StubActivity` 的 `Intent`，并将原 `Intent` 存入 extra。

1. 启动与还原:

- 系统正常启动 `StubActivity`。
- `StubActivity` 在 `onCreate` 中，解析出插件 Activity 的类名。
- 使用 `DexClassLoader` 反射创建插件 Activity 实例。
- 将自定义 `Resources` 等注入插件 Activity 的 `Context`。
- 手动调用插件 Activity 的 `onCreate()` 等生命周期方法。

至此，插件 Activity 的界面就显示出来了，但它实际上
是运行在 `StubActivity` 的"壳"里。

知名开源项目参考

从零开始构建一个完整的沙箱框架非常困难，以下项目是极佳的学习资源：

- VirtualApp: 最著名的 Android 沙箱项目之一，代码结构清晰，是学习原理的绝佳范例。
- DroidPlugin: 由 360 开发的早期沙箱项目，对四大组件的支持非常完整。

挑战与局限

- 兼容性: Android 版本每次大更新, 大量系统服务内部实现会改变, 需要持续适配。
- 复杂性: 需要处理四大组件、文件系统、Content Provider、系统广播等方方面面的虚拟化。
- Native Code: 对包含 JNI/Native 代码的 App 支持起来更复杂, 可能需要对 so 文件的加载和符号解析进行 Hook。
- 厂商 ROM: 不同手机厂商对 Android 系统的魔改, 也可能导致沙箱在某些设备上失效。

[A02] AOSP 与系统定制

A02: AOSP 与 Android 系统裁剪

Android 开源项目 (AOSP) 是 Android 操作系统的开源基础。能够编译和修改 AOSP 是进行深度系统级定制、安全研究和 ROM 开发的核心技能。本节将介绍 AOSP 的基本概念、编译流程以及常见的系统裁剪技术。

1. AOSP 基础与编译

a) AOSP 源码同步

编译 AOSP 的第一步是获取其庞大的源代码树。

1. 环境准备:

- 一个强大的 Linux 构建服务器（推荐 Ubuntu LTS），至少需要 16GB RAM 和 300GB 的可用磁盘空间。
- 安装必要的依赖包，如 `git`, `curl`, `python`, `Java SDK` 等。

2. 获取 Repo 工具: `Repo` 是 Google 开发的、基于 Git 的代码库管理工具，用于管理 AOSP 中数百个不同的 Git 仓库。

```
# 下载 Repo 工具
mkdir -p ~/.bin
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/.bin/repo
chmod a+x ~/.bin/repo
export PATH=~/.bin:$PATH
```

```
# 创建工作目录  
mkdir aosp && cd aosp  
  
# 初始化源码仓库, 指定分支 (例如 android-12.0.0_r1)  
repo init -u https://android.googlesource.com/platform/manifest -b android-12.0.0_r1  
--depth=1  
  
# 开始同步源码 (这将是一个漫长的过程)  
repo sync -c -j8
```

- `--depth=1`: 只同步最新的 commit, 大幅减少下载量。
- `-c`: 只同步当前分支。
- `-j8`: 使用 8 个线程并行同步。

b) 编译流程

1. 设置环境:

```
source build/envsetup.sh
```

1. 选择目标 (Lunch):

```
lunch aosp_arm64-eng
```

- `aosp_arm64`: 目标设备架构 (64 位 ARM)。
- `eng`: 构建变体 (Engineering), 包含最多的调试工具, 权限为 root, 适合开发和逆向。其他还有 `user` (发布版) 和 `userdebug` (带 root 和调试功能的用户版)。

1. 开始编译 (Make):

```
m
```

1. 编译产物:

编译完成后，所有的系统镜像文件都存放在 `out/target/product/<device_name>/` 目录下，主要包括：

- `system.img`：系统分区镜像。
- `vendor.img`：厂商分区镜像。
- `boot.img`：启动分区镜像，包含内核和 ramdisk。
- `userdata.img`：用户数据分区镜像。

2. 系统裁剪与定制技术

拥有了编译 AOSP 的能力后，你就可以对系统进行任意的修改。

a) 预置与删除 App

- 路径: App 通常定义在 `packages/apps/` 目录下。
- 修改 `PRODUCT_PACKAGES`：在特定设备的 `device.mk` 文件中（例如 `device/<vendor>/<device_name>/device.mk`），有一个名为 `PRODUCT_PACKAGES` 的变量。
- 增加 App: 将你想要预置的 App 的模块名添加到这个列表中。
- 删除 App: 从这个列表中移除你不想要的系统 App（如 `Calendar`, `Camera2`）的模块名。

b) 修改 Framework 层

这是更深度的定制，可以改变 Android 系统的核心行为。

- 路径: Framework 核心代码位于 `frameworks/base/`。
- 示例：修改状态栏逻辑：
 1. 找到负责状态栏管理的 `SystemUI` App (`frameworks/base/packages/SystemUI/`)。
 2. 修改其中的 Java 或 XML 文件，例如，改变时钟的显示格式或电池图标。
 3. 重新编译 `SystemUI` 模块：`m SystemUI`。
 4. 只编译模块并生成新的 `system.img`：`m snod` (`make systemimage-nodeps`)。

c) 定制内核 (Kernel)

AOSP 默认不包含内核源码。你需要从 Google 的内核源码仓库或设备厂商的开源站点单独下载内核源码，并进行编译。

1. 获取内核源码:

```
git clone https://android.googlesource.com/kernel/common.git
```

1. 配置与编译:

```
# 使用与 AOSP 匹配的交叉编译工具链  
export CROSS_COMPILE=.../aarch64-linux-android-4.9/bin/aarch64-linux-android-  
  
# 配置内核  
make defconfig  
  
# 编译内核镜像  
make
```

d) 制作完整的自定义 ROM

一个完整的自定义 ROM（如 LineageOS）的制作过程，就是上述所有技术的综合应用：

1. 同步 AOSP 基础代码。
2. 集成特定设备的驱动和配置文件（Device Tree）。
3. 修改 Framework，添加自定义功能（如高级重启菜单）。
4. 移除或替换系统 App。
5. 集成定制的内核。
6. 编译并打包成一个可供用户刷写的 `zip` 文件。

3. Android Linker 与 SO 加载原理

Linker 架构与工作原理

Android 系统使用动态链接器（`/system/bin/linker` 或 `/system/bin/linker64`）来加载和链接共享库（.so 文件）。

系统架构

```
Runtime.loadLibrary()
    ↓
DexPathList.loadLibrary()
    ↓
nativeLoad() [JNI]
    ↓
android_dlopen_ext()
    ↓
do_dlopen() [linker]
    ↓
find_library_internal()
    ↓
load_library() → link_image()
```

核心函数分析

`find_library_internal` - 查找库:

```
static soinfo* find_library_internal(android_namespace_t* ns,
                                    const char* name,
                                    int rtld_flags,
                                    const android_dlexinfo* extinfo,
                                    soinfo* needed_by) {
    // 1. 检查是否已加载
    soinfo* si = find_loaded_library_by_soname(ns, name);
    if (si != nullptr) {
        return si;
    }

    // 2. 在命名空间中搜索
    std::string realpath;
    if (!find_library_in_namespace(ns, name, &realpath)) {
        return nullptr;
    }

    // 3. 加载库文件
    return load_library(ns, realpath.c_str(), rtld_flags, extinfo, needed_by);
}
```

load_library - 加载库:

```
static soinfo* load_library(android_namespace_t* ns,
                           const char* name,
                           int rtld_flags,
                           const android_dlexinfo* extinfo,
                           soinfo* needed_by) {
    // 1. 打开 ELF 文件
    int fd = open(name, O_RDONLY | O_CLOEXEC);

    // 2. 解析 ELF 头
    ElfReader elf_reader(name, fd, file_offset, file_size);
    if (!elf_reader.Load(extinfo)) {
        return nullptr;
    }

    // 3. 创建 soinfo 结构
    soinfo* si = soinfo_alloc(ns, realpath, &file_stat, rtld_flags, extinfo);

    // 4. 映射内存段
    if (!si->prelink_image()) {
        return nullptr;
    }

    return si;
}
```

link_image - 链接镜像:

```
bool soinfo::link_image(const soinfo_list_t& global_group,
                       const soinfo_list_t& local_group,
                       const android_dlextinfo* extinfo) {
    // 1. 解析动态段
    if (!phdr_table_get_dynamic_section(phdr, phnum, load_bias, &dynamic,
                                         &dynamic_flags)) {
        return false;
    }

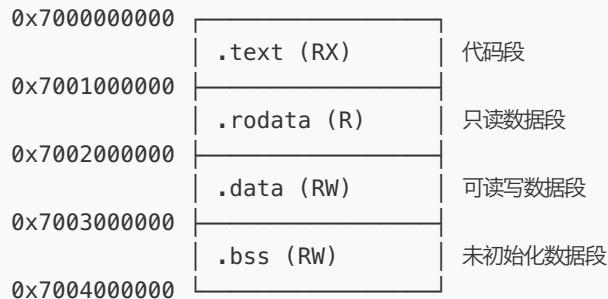
    // 2. 处理依赖库
    for (ElfW(Dyn)* d = dynamic; d->d_tag != DT_NULL; ++d) {
        if (d->d_tag == DT_NEEDED) {
            const char* library_name = get_string(d->d_un.d_val);
            soinfo* lsi = find_library(library_name, ...);
        }
    }

    // 3. 重定位处理
    if (!relocate(global_group, local_group)) {
        return false;
    }

    // 4. 调用构造函数
    call_constructors();

    return true;
}
```

SO 内存布局



内存保护设置

```
int phdr_table_protect_segments(const ElfW(Phdr)* phdr_table,
                                size_t phdr_count,
                                ElfW(Addr) load_bias) {
    for (size_t i = 0; i < phdr_count; ++i) {
        const ElfW(Phdr)* phdr = &phdr_table[i];
        if (phdr->p_type != PT_LOAD) continue;

        int prot = PFLAGS_TO_PROT(phdr->p_flags);
        if (mprotect(reinterpret_cast<void*>(seg_page_start + load_bias),
                     seg_page_end - seg_page_start, prot) < 0) {
            return -1;
        }
    }
    return 0;
}
```

4. 反调试与检测技术

init 函数中的反调试

```
__attribute__((constructor))
void anti_debug_check() {
    // 检测 Frida
    if (access("/data/local/tmp/frida-server", F_OK) == 0) {
        _exit(1);
    }

    // 检测调试器
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1) {
        _exit(1);
    }

    // 检测虚拟机环境
    check_emulator_files();
}
```

符号表检测

```
void check_loaded_symbols() {
    void* handle = dlopen(NULL, RTLD_NOW);

    // 检测 Frida 符号
    if (dlsym(handle, "frida_agent_main") != NULL) {
        _exit(1);
    }

    // 检测 Xposed 符号
    if (dlsym(handle, "xposed_bridge") != NULL) {
        _exit(1);
    }
}
```

代码完整性校验

```
void check_code_integrity() {
    Dl_info info;
    dladdr((void*)check_code_integrity, &info);

    // 计算代码段哈希
    uint32_t current_hash = calculate_hash(info.dli_fbase, TEXT_SIZE);
    if (current_hash != EXPECTED_HASH) {
        _exit(1);
    }
}
```

5. Frida 绕过技术

绕过 ptrace 检测

```
var ptrace = Module.findExportByName("libc.so", "ptrace");
if (ptrace) {
    Interceptor.attach(ptrace, {
        onEnter: function(args) {
            var request = args[0].toInt32();
            if (request === 0) { // PTTRACE_TRACEME
                console.log("[+] ptrace(PTTRACE_TRACEME) blocked");
                args[0] = ptr(-1); // 修改参数使其失败
            }
        },
        onLeave: function(retval) {
            retval.replace(ptr(0)); // 返回成功
        }
    });
}
```

隐藏 Frida 符号

```
var dlsym = Module.findExportByName("libdl.so", "dlsym");
Interceptor.attach(dlsym, {
    onEnter: function(args) {
        this.symbol_name = args[1].readCString();
    },
    onLeave: function(retval) {
        var blocked_symbols = ["frida_agent_main", "xposed_bridge"];
        if (blocked_symbols.includes(this.symbol_name)) {
            console.log("[+] Hiding symbol: " + this.symbol_name);
            retval.replace(ptr(0));
        }
    }
});
```

绕过 mprotect 检测

```
var mprotect = Module.findExportByName("libc.so", "mprotect");
Interceptor.attach(mprotect, {
    onEnter: function(args) {
        var addr = args[0];
        var size = args[1].toInt32();
        var prot = args[2].toInt32();

        console.log("[+] mprotect called: " + addr + ", size: " + size + ", prot: " +
prot);

        // 防止移除执行权限
        if ((prot & 0x4) == 0) { // PROT_EXEC
            args[2] = ptr(prot | 0x4);
        }
    }
});
```

隐藏敏感路径

```
var access = Module.findExportByName("libc.so", "access");
Interceptor.attach(access, {
    onEnter: function(args) {
        var path = args[0].readCString();
        var sensitive_paths = [
            "/data/local/tmp/frida-server",
            "/system/xbin/su",
            "/system/app/Superuser.apk"
        ];

        if (sensitive_paths.some(p => path.includes(p))) {
            console.log("[+] Blocking access to: " + path);
            args[0] = Memory.allocUtf8String("/non/existent/path");
        }
    }
});
```

6. init_array 注入技术

Python 脚本注入

```
from elftools.elf.elffile import ELFFile

def inject_init_array(elf_path, hook_function_addr):
    with open(elf_path, 'r+b') as f:
        elf = ELFFile(f)

        # 查找 .init_array 段
        init_array = elf.get_section_by_name('.init_array')
        if init_array:
            # 在现有函数指针后添加新函数地址
            f.seek(init_array['sh_offset'] + init_array['sh_size'])
            f.write(hook_function_addr.to_bytes(8, 'little'))
```

C 代码运行时注入

```
void inject_init_array_runtime() {
    // 1. 找到目标SO 加载基址
    void* base_addr = dlopen("target.so", RTLD_NOLOAD);

    // 2. 解析 ELF 头找到 .init_array 段
    ElfW(Ehdr)* ehdr = (ElfW(Ehdr)* )base_addr;
    ElfW(Shdr)* shdr = (ElfW(Shdr)* )((char*)base_addr + ehdr->e_shoff);

    // 3. 修改内存保护
    mprotect(init_array_addr, init_array_size, PROT_READ | PROT_WRITE);

    // 4. 添加函数指针
    *(void**)(init_array_addr + init_array_size) = target_function;

    // 5. 恢复保护
    mprotect(init_array_addr, init_array_size, PROT_READ);
}
```

7. 综合反调试检测

```
class AntiDebugChecker {  
private:  
    static bool check_debugger_presence() {  
        return ptrace(PTRACE_TRACEME, 0, 1, 0) == -1;  
    }  
  
    static bool check_frida_artifacts() {  
        const char* frida_files[] = {  
            "/data/local/tmp/frida-server",  
            "/data/local/tmp/frida-agent-64.so"  
        };  
  
        for (auto file : frida_files) {  
            if (access(file, F_OK) == 0) return true;  
        }  
        return false;  
    }  
  
    static bool check_memory_maps() {  
        FILE* fp = fopen("/proc/self/maps", "r");  
        char line[512];  
        while (fgets(line, sizeof(line), fp)) {  
            if (strstr(line, "frida") || strstr(line, "gum-js-loop")) {  
                fclose(fp);  
                return true;  
            }  
        }  
        fclose(fp);  
        return false;  
    }  
  
public:  
    static void comprehensive_check() {  
        if (check_debugger_presence() ||  
            check_frida_artifacts() ||  
            check_memory_maps()) {  
            // 执行对抗措施  
            execute_countermeasures();  
        }  
    }  
};
```

定时检测线程

```
void start_anti_debug_thread() {
    std::thread([]{
        while (true) {
            std::this_thread::sleep_for(std::chrono::seconds(5));
            AntiDebugChecker::comprehensive_check();
        }
    }).detach();
}
```

总结

掌握 AOSP 编译和系统定制技术的意义：

- 深入理解 Android Framework 的工作原理，为 Hook 和逆向提供更底层的视角。
- 通过修改系统来绕过应用层的反分析技术，实现“降维打击”。

[A03] AOSP 深度改机

A03: 基于 AOSP 的深度改机技术指南

在 Android 安全和逆向工程领域，"改机"指的是修改设备的各种硬件和软件标识符，以绕过应用程序的安全检测或实现隐私保护。虽然使用 Xposed 或 Frida 等 Hook 框架可以在应用层实现改机，但这些方法容易被检测。基于 AOSP (Android Open Source Project) 源码进行修改，是从系统层面伪造设备指纹的终极手段，因为 App 获取到的信息是由系统本身"真实"地提供的。

本文旨在提供一个关于如何通过修改 AOSP 源码来实现深度改机的技术框架和思路。

目录

1. 核心思想：应用层 Hook vs. 系统层修改
2. 准备工作
3. 关键参数定位与修改
 - Build Info (build.prop)
 - 硬件参数 (IMEI, MAC, Android ID)
 - 系统属性 (System Properties)
 - 内核参数 (Serial Number)
 - Build Info (build.prop)
 - 硬件参数 (IMEI, MAC, Android ID)
 - 系统属性 (System Properties)

- 内核参数 (Serial Number)

1. 编译与刷机

2. 优势与挑战

核心思想：应用层 Hook vs. 系统层修改

特性	应用层 Hook (Xposed/Frida)	AOSP 系统层修改
原理	在 App 运行时，拦截 API 调用，返回伪造结果。	直接修改 Android 框架层源码，使 API 本身就返回伪造的值。
效果	较好，但可被检测。	极好，效果彻底。
检测难度	容易被反 Hook、反调试技术检测到。	极难被检测，因为对 App 来说系统行为是"原生"的。
实现难度	相对较低，只需编写 Hook 脚本。	非常高，需要编译整个 Android 系统。
适用性	通用性强，适用于大多数设备。	通常只适用于 AOSP 支持良好的设备（如 Google Pixel）。

结论：AOSP 改机的本质是构建一个“出厂设置”就是伪装状态的自定义操作系统。

准备工作

1. 硬件要求：

- 一台高性能的 PC (至少 16GB RAM, 推荐 32GB 或更高)。
- 大容量高速硬盘 (SSD, 至少 500GB 可用空间)。

-
- 一台受 AOSP 官方支持的设备（如 Google Pixel 系列），用于刷机验证。
 - 一台高性能的 PC（至少 16GB RAM，推荐 32GB 或更高）。
 - 大容量高速硬盘（SSD，至少 500GB 可用空间）。
 - 一台受 AOSP 官方支持的设备（如 Google Pixel 系列），用于刷机验证。

1. 软件环境:

- Linux 操作系统（推荐 Ubuntu LTS 版本）。
- 熟悉 Android 编译环境，安装好 `repo` 和所有必需的依赖库。
- Linux 操作系统（推荐 Ubuntu LTS 版本）。
- 熟悉 Android 编译环境，安装好 `repo` 和所有必需的依赖库。

1. AOSP 源码:

- 根据你的目标设备和 Android 版本，初始化并同步对应的 AOSP 源码仓库。
 - 根据你的目标设备和 Android 版本，初始化并同步对应的 AOSP 源码仓库。
-

关键参数定位与修改

Build Info (build.prop)

这些是描述设备型号、品牌、制造商等最基础的信息。

- 定位: 这些值通常定义在 `device/` 目录下的特定于设备的 `*.mk` makefile 文件中，或者在 `build/make/target/product/` 下的通用产品定义文件中。
- 修改示例:
 - 打开 `device/<vendor>/<product_name>/device.mk` 或类似文件。
 - 找到并修改以下变量:

```
PRODUCT_MODEL := Pixel 8 Pro
PRODUCT_BRAND := Google
PRODUCT_NAME := my_custom_device
PRODUCT_DEVICE := generic
PRODUCT_MANUFACTURER := MyCompany
```

硬件参数 (IMEI, MAC, android ID)

这些是更敏感、更核心的设备标识符。修改它们需要深入到 Framework 层的 Java 代码和 JNI。

* **IMEI (Telephony)**:

* **定位**: `frameworks/opt/telephony/src/java/com/android/internal/telephony/Phone.java` 或相关的 `*SubInfo.java` 文件。

* **修改思路**: 找到 `getImei()` 或类似方法，在其中硬编码或返回一个动态生成的伪造 IMEI。

* **定位**: `frameworks/opt/telephony/src/java/com/android/internal/telephony/Phone.java` 或相关的 `*SubInfo.java` 文件。

* **修改思路**: 找到 `getImei()` 或类似方法，在其中硬编码或返回一个动态生成的伪造 IMEI。

* **MAC Address (Wi-Fi)**:

* **定位**: `frameworks/base/wifi/java/android/net/wifi/WifiInfo.java`。

* **修改思路**: 找到 `getMacAddress()` 方法。注意，在高版本 Android 中，该方法可能返回一个固定的、非真实的 MAC 地址。需要找到其更底层的实现，可能在 `wpa_supplicant` 或 Wi-Fi 驱动的 JNI 接口中。

* **定位**: `frameworks/base/wifi/java/android/net/wifi/WifiInfo.java`。

* **修改思路**: 找到 `getMacAddress()` 方法。注意，在高版本 Android 中，该方法可能返回一个固定的、非真实的 MAC 地址。需要找到其更底层的实现，可能在 `wpa_supplicant` 或 Wi-Fi 驱动的 JNI 接口中。

* **Android ID**:

* **定位**: `frameworks/base/services/core/java/com/android/server/pm/Settings.java` 中的 `getStringForUser()` 方法，结合 `android.provider.Settings.Secure.ANDROID_ID` 的实现。

* **修改思路**: 找到生成和存储 Android ID 的逻辑，将其替换为返回一个固定的或每次启动都随机生成的值。

* **定位**: `frameworks/base/services/core/java/com/android/server/pm/Settings.java` 中的 `getStringForUser()` 方法，结合 `android.provider.Settings.Secure.ANDROID_ID` 的实现。

* **修改思路**: 找到生成和存储 Android ID 的逻辑，将其替换为返回一个固定的或每次启动都随机生成的值。

系统属性 (System Properties)

App 通过 `android.os.SystemProperties.get()` 获取各种系统属性。

* **定位**: `frameworks/base/core/java/android/os/SystemProperties.java` 及其对应的 JNI 实现 `frameworks/base/core/jni/android_os_SystemProperties.cpp`。

* **修改思路**: 直接在 `SystemProperties.cpp` 的 `native_get` 方法中进行拦截。判断传入的属性名，如果是目标属性（如 `ro.serialno`），则返回一个伪造的值，否则执行原始逻辑。

内核参数 (Serial Number)

一些底层信息（如 CPU 序列号）直接由 Linux 内核通过 `/proc` 文件系统暴露。

* **定位**: 内核源码中的 `arch/<arch>/kernel/setup.c` 或相关驱动文件。

* **修改思路**:

1. 下载与 AOSP 版本匹配的内核源码。
2. 找到向 `/proc/cpuinfo` 或 `/proc/serial` 等文件写入信息的代码。
3. 修改这部分逻辑，使其输出伪造的信息。
4. 重新编译内核 (`boot.img`)。

1. 下载与 AOSP 版本匹配的内核源码。

2. 找到向 `/proc/cpuinfo` 或 `/proc/serial` 等文件写入信息的代码。

3. 修改这部分逻辑，使其输出伪造的信息。

4. 重新编译内核 (`boot.img`)。

编译与刷机

1. **设置环境**: `source build/envsetup.sh`

2. **选择目标**: `lunch aosp_<device_name>-userdebug` (例如 `lunch aosp_husky-userdebug` 对应 Pixel 8 Pro)

3. **开始编译**: `make -j\$(nproc)` (这会花费数小时)

4. **刷机**:

* 将设备置于 `fastboot` 模式。

* 执行 `fastboot flashall -w`，这将刷写所有编译生成的镜像 (`system.img`, `boot.img`, `vendor.img` 等)。

* 将设备置于 `fastboot` 模式。

* 执行 `fastboot flashall -w`，这将刷写所有编译生成的镜像 (`system.img`, `boot.img`, `vendor.img` 等)。

优势与挑战

优势

* **彻底性**: 从系统根源上改变设备指纹，几乎无法被应用层技术检测。

* **稳定性**: 不会像 Hook 框架那样因为应用更新或加固而失效。

* **性能好**: 没有额外的 Hook 开销，所有修改都是原生代码。

挑战

* **技术门槛极高**: 需要深入理解 AOSP 源码结构、编译系统和 Linux 内核。

* **时间成本高**: 全量编译一次 AOSP 通常需要数小时。

* **设备限制**: 强依赖于有良好 AOSP 支持和开放驱动的设备。

* **维护困难**: 每次 Android 版本更新, 都需要重新进行源码适配和修改。

[A04] 最小化 Android RootFS

A04: 构建最小化 Android 系统 (RootFS) 指南

构建一个完整的 AOSP (Android Open Source Project) 耗时巨大且对硬件要求苛刻。而构建一个最小化的 Android RootFS (Root File System) 是一个能让我们深刻理解 Android 启动流程和核心组件的绝佳实践。其目标是创建一个仅包含最基本组件、能够引导 Linux 内核并最终启动一个交互式 Shell 的系统。

本文将指导你完成这一过程，主要使用 QEMU 作为目标平台。

目录

- 构建最小化 android 系统 (RootFS) 指南
 - 本文将指导你完成这一过程，主要使用 QEMU 作为目标平台。
 - 目录
 - 核心概念与启动流程
 - 最小系统的核心组件
 - 构建步骤详解
 - Step 1: 准备环境与工具链
 - Step 2: 获取并编译 Linux 内核
 - Step 3: 构建最小化 RootFS
 - Step 4: 打包并运行
 - 从 Shell 到 Zygote: 下一步是什么？

核心概念与启动流程

1. Bootloader: 设备上电后执行的第一段代码, 负责初始化硬件并加载 Linux 内核到内存。
2. Kernel: 内核被加载后, 开始初始化各种驱动、内存管理等, 然后挂载一个临时的根文件系统 (ramdisk)。
3. `init` 进程: 内核在用户空间启动的第一个进程, 其 PID 为 1。它是所有其他用户空间进程的祖先。
4. `init.rc`: `init` 进程会解析这个配置文件, 根据其中的指令执行动作, 如挂载文件系统、设置系统属性、启动服务等。

我们的目标就是创建一个极简的 RootFS, 其中包含 `init` 程序和一个能被它启动的 Shell。

最小系统的核心组件

一个能启动到 Shell 的最小 Android 系统, 必须包含以下组件:

- Linux Kernel: 操作系统的核心。
- `init`: 用户空间的守护神, 来自 AOSP 源码 `system/core/init`。
- C 库: `libc.so` (C 标准库), `libm.so` (数学库)。所有原生程序都依赖它。
- 动态链接器: `linker` 或 `linker64`, 用于加载 `.so` 动态库。
- Shell: `sh`, 我们的交互界面, 通常由 `toybox` 或 `toolbox` 提供。
- `init.rc`: 一个最简单的配置文件。
- 基本目录结构: `/dev`, `/proc`, `/sys`, `/system/bin`。

构建步骤详解

Step 1: 准备环境与工具链

你需要一个 Linux 环境（如 Ubuntu）和用于交叉编译的工具链。最简单的方法是从 AOSP 预编译库中获取。

```
# Download AOSP prebuilt aarch64 (ARM64) Toolchain
git clone https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/aarch64/
aarch64-linux-android-4.9

# Add Toolchain Path to Environment variables
export PATH=$(pwd)/aarch64-linux-android-4.9/bin:$PATH
export CROSS_COMPILE=aarch64-linux-android-
```

Step 2: 获取并编译 Linux 内核

```
git clone https://android.googlesource.com/kernel/common.git
cd common

# Switch to a Stable branch
git checkout android-xxxx

# 配置内核
export ARCH=arm64
make defconfig

# 编译内核
make -j$(nproc)

# Compile Success. After this, Image.gz will be generated in arch/arm64/boot/
directory
```

Step 3: 构建最小化 RootFS

```
mkdir -p my_rootfs/{dev,proc,sys,system/bin,system/lib64}
cd my_rootfs
```

这一步比较复杂，因为需要从完整的 AOSP 源码中单独编译。一个简化的方法是直接从一个现有的 Android 系统或 AOSP 编译产物中提取这些预编译好的二进制文件。

- 从 AOSP 编译产物 `out/target/product/<device>/system/` 中找到以下文件：

- `bin/linker64` -> 复制到 `my_rootfs/system/bin/`
- `bin/init` -> 复制到 `my_rootfs/`
- `bin/toybox` -> 复制到 `my_rootfs/system/bin/`
- `lib64/libc.so`, `lib64/libm.so` -> 复制到 `my_rootfs/system/lib64/`

- 为 `toybox` 创建各种命令的软链接：

```
cd my_rootfs/system/bin
for cmd in $(./toybox); do
    ln -s toybox $cmd
done
cd ../../
```

在 `my_rootfs/` 目录下创建一个 `init.rc` 文件，内容如下：

```
# init.rc for minimal android

on early-init
    mount tmpfs /dev
    mkdir /dev/pts
    mount devpts devpts /dev/pts
    mount proc proc /proc
    mount sysfs sysfs /sys

on init
    export PATH /system/bin
    export LD_LIBRARY_PATH /system/lib64

on post-fs
    # In a real system, we would mount /data, /cache, etc.
    # Here we just start the shell.

service shell /system/bin/sh
    class core
    console
    disabled
    user shell
    group shell
    seclabel u:r:shell:s0

on property:sys.boot_completed=1
    start shell
```

Step 4: 打包并运行

1. 打包 RootFS: 我们需要将 `my_rootfs` 目录打包成一个 `cpio` 归档，并用 `gzip` 压缩，作为内核的 `initramfs`。

```
cd my_rootfs
find . | cpio -o -H newc | gzip > ../rootfs.cpio.gz
cd ..
```

1. 运行 QEMU: 确保 `common/arch/arm64/boot/Image.gz` 和 `rootfs.cpio.gz` 在当前目录下。

```
qemu-system-aarch64 -M virt -cpu cortex-a57 -m 2048 -kernel common/arch/arm64/boot/Image.gz -initrd rootfs.cpio.gz -nographic -append "console=ttyAMA0"
```

从 Shell 到 Zygote：下一步是什么？

我们已经有了一个最小的 Linux 环境，但它还不是"Android"。要让它成为 Android，还需要以下关键步骤：

1. 启动 `servicemanager`：编译并运行它，它是 Android Binder IPC 机制的核心。
2. 启动 Zygote：编译 `app_process` 并通过 `init.rc` 启动它。Zygote 会预加载 Android 框架的核心类（`framework.jar`）并监听一个 socket，等待孵化新的 App 进程。
3. 启动 `system_server`：Zygote 启动的第一个 Java 进程，它会创建所有的 Android 系统服务（AMS, WMS, PMS 等）。

完成这些后，系统才能真正地运行 Android 应用。但这已经超出了"最小化 RootFS"的范畴，进入了完整的系统移植和开发领域。

[A05] SO 反调试与混淆

A05: SO 文件反调试与字符串混淆技术

在 Android Native 层安全对抗中，SO 文件是实现高强度保护的重要载体。通过 init_array 机制、字符串混淆和反调试技术的组合使用，可以显著提高逆向分析的难度。本文将深入分析这些技术的实现原理及对应的分析绕过方法。

目录

-
- 1. [init_array 调用流程原理](#)
 - 2. [字符串混淆技术](#)
 - 3. [反调试技术实现](#)
 - 4. [分析与绕过方法](#)
 - 5. [高级防护策略](#)
-

1. init_array 调用流程原理

1.1 ELF 加载与 init_array 执行时机

```
// linker 中 call_constructors 的简化实现
void soinfo::call_constructors() {
    // 1. 首先调用 DT_INIT 初始化函数
    if (init_func_ != nullptr) {
        init_func_();
    }

    // 2. 然后遍历 .init_array 段的函数指针
    if (init_array_ != nullptr) {
        for (size_t i = 0; i < init_array_count_; ++i) {
            // 调用每个构造函数
            ((void (*)())init_array_[i])();
        }
    }
}
```

1.2 完整调用链路

```
System.loadLibrary("target")
    ↓
nativeLoad() [art/runtime/native/java_lang_Runtime.cc]
    ↓
android_dlopen_ext() [bionic/libdl/libdl.cpp]
    ↓
do_dlopen() [bionic/linker/linker.cpp]
    ↓
find_library() → load_library() → link_image()
    ↓
call_constructors() → init_array 函数执行
```

1.3 查看 init_array 信息

```
# 使用 readelf 查看动态段
readelf -d target.so | grep INIT

# 使用 objdump 分析
objdump -s -j .init_array target.so
```

1.4 init_array 结构

```
// init_array 结构信息
typedef struct {
    Elf64_Addr *init_array;      // 函数指针数组
    size_t init_array_count;     // 数组大小
} init_array_info;

// 反调试函数声明
__attribute__((constructor))
void anti_debug_init() {
    // 反调试逻辑
}

// 编译后会在 .init_array 段生成函数指针
```

2. 字符串混淆技术

2.1 XOR 加密字符串

```
// 字符串加密宏定义
#define ENCRYPT_STRING(str) encrypt_string_xor(str, sizeof(str)-1, 0xAA)

constexpr char* encrypt_string_xor(const char* str, size_t len, char key) {
    static char encrypted[256];
    for (size_t i = 0; i < len; i++) {
        encrypted[i] = str[i] ^ key;
    }
    encrypted[len] = '\0';
    return encrypted;
}

// 使用示例
void check_frida() {
    // 原始字符串: "/data/local/tmp/frida-server"
    const char* encrypted = "\xc4\xae\x8\x8\xe4\xe6\xe8\xe0\xe4\xe6\xe4";

    char decrypted[256];
    decrypt_string(encrypted, decrypted, strlen(encrypted), 0xAA);

    if (access(decrypted, F_OK) == 0) {
        exit(1);
    }
}
```

2.2 AES 加密字符串

```
class String0bfuscator {
private:
    static constexpr uint8_t AES_KEY[16] = {
        0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
        0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c
    };

    static void aes_decrypt(const uint8_t* encrypted, uint8_t* decrypted, size_t len)
    {
        AES_KEY aes_key;
        AES_set_decrypt_key(AES_KEY, 128, &aes_key);

        for (size_t i = 0; i < len; i += 16) {
            AES_decrypt(encrypted + i, decrypted + i, &aes_key);
        }
    }

public:
    static std::string decrypt_string(const uint8_t* encrypted_data, size_t len) {
        std::vector<uint8_t> decrypted(len);
        aes_decrypt(encrypted_data, decrypted.data(), len);

        // 移除 padding
        size_t actual_len = len;
        while (actual_len > 0 && decrypted[actual_len - 1] == 0) {
            actual_len--;
        }

        return std::string(reinterpret_cast<char*>(decrypted.data()), actual_len);
    }
};

// 使用加密字符串
void advanced_anti_debug() {
    // 加密的 "/proc/self/status" 字符串
    const uint8_t encrypted_proc_status[] = {
        0x8a, 0x2d, 0x5e, 0x1f, 0x9b, 0x7c, 0x85, 0xa3,
        0x4e, 0x92, 0x67, 0xc1, 0x55, 0x98, 0x33, 0xa
    };

    std::string proc_status = String0bfuscator::decrypt_string(
        encrypted_proc_status, sizeof(encrypted_proc_status)
    );

    check_debugger_via_status(proc_status.c_str());
}
```

2.3 栈上动态构造字符串

```
void construct_string_on_stack() {
    char target_path[64];

    // 分段构造字符串
    strcpy(target_path, "/data/");
    strcat(target_path, "local/");
    strcat(target_path, "tmp/");
    strcat(target_path, "frida-");
    strcat(target_path, "server");

    if (access(target_path, F_OK) == 0) {
        exit(1);
    }

    // 清理栈上敏感字符串
    memset(target_path, 0, sizeof(target_path));
}
```

2.4 动态字符串构建器

```
class DynamicStringBuilder {
private:
    std::vector<std::string> fragments;

public:
    void add_fragment(const char* encrypted, size_t len, uint8_t key) {
        std::string decrypted;
        for (size_t i = 0; i < len; i++) {
            decrypted += static_cast<char>(encrypted[i] ^ key);
        }
        fragments.push_back(decrypted);
    }

    std::string build() {
        std::string result;
        for (const auto& fragment : fragments) {
            result += fragment;
        }

        // 立即清理 fragments
        fragments.clear();

        return result;
    }
};

void dynamic_string_detection() {
    DynamicStringBuilder builder;

    // 分段加密字符串片段
    const char frag1[] = {0x8f, 0x9e, 0x9a, 0x9a, 0x8f}; // "/data"
    const char frag2[] = {0x8f, 0x93, 0x91, 0x9d, 0x9e, 0x93}; // "/local"
    const char frag3[] = {0x8f, 0x9a, 0x94, 0x92}; // "/tmp"

    builder.add_fragment(frag1, 5, 0xEE);
    builder.add_fragment(frag2, 6, 0xEE);
    builder.add_fragment(frag3, 4, 0xEE);

    std::string path = builder.build();

    // 使用构造的路径进行检测
    perform_detection(path.c_str());
}
```

3. 反调试技术实现

3.1 init_array 中的反调试

```
// 在 .init_array 中执行反调试函数
__attribute__((constructor(101))) // 指定优先级
void init_anti_debug_level1() {
    // 1. ptrace 自身保护
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1) {
        _exit(1);
    }

    // 2. 检测调试器进程
    check_debugger_processes();

    // 3. 检测 Frida 文件
    check_frida_artifacts();
}

__attribute__((constructor(102)))
void init_anti_debug_level2() {
    // 4. 检测内存映射
    check_suspicious_mappings();

    // 5. 检测 hook 痕迹
    check_hook_signatures();

    // 6. 时间检测
    timing_attack_detection();
}

void check_debugger_processes() {
    const char* debugger_names[] = {
        "gdb", "lldb", "strace", "ida", "x64dbg"
    };

    for (const char* name : debugger_names) {
        if (process_exists(name)) {
            execute_anti_debug_response();
        }
    }
}

void check_frida_artifacts() {
    const char* frida_indicators[] = {
        "/data/local/tmp/frida-server",
        "/data/local/tmp/frida-agent-64.so",
        "/system/lib64/libfrida-gum.so"
    };

    for (const char* indicator : frida_indicators) {
        if (file_exists(indicator)) {
            execute_anti_debug_response();
        }
    }
}
```

3.2 内存映射检测

```
__attribute__((constructor(103)))
void init_memory_protection() {
    // 检测代码段完整性
    verify_code_integrity();

    // 检测异常向量表
    check_exception_handlers();

    // 设置内存保护
    setup_memory_protection();
}

void check_suspicious_mappings() {
    FILE* maps = fopen("/proc/self/maps", "r");
    char line[512];

    while (fgets(line, sizeof(line), maps)) {
        // 检测可疑库映射
        if (strstr(line, "frida") ||
            strstr(line, "gum-js-loop") ||
            strstr(line, "xposed")) {
            fclose(maps);
            execute_anti_debug_response();
        }

        // 检测可疑权限组合
        if (strstr(line, "rwxp")) { // 可读写执行页面
            analyze_rwx_mapping(line);
        }
    }

    fclose(maps);
}

void verify_code_integrity() {
    // 计算代码段哈希值
    Dl_info info;
    dladdr((void*)verify_code_integrity, &info);

    const char* base = (const char*)info.dli_fbase;
    size_t text_size = get_text_section_size(base);

    uint32_t current_hash = calculate_crc32(base, text_size);
    uint32_t expected_hash = get_expected_hash();

    if (current_hash != expected_hash) {
        // 代码被修改，执行对抗措施
        code_tampering_detected();
    }
}
```

3.3 时间检测

```
void init_timing_checks() {
    // 启动定时器检测
    start_timing_monitor();

    // 检测单步执行
    detect_single_stepping();
}

void detect_single_stepping() {
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    // 执行一些简单操作
    volatile int dummy = 0;
    for (int i = 0; i < 1000; i++) {
        dummy += i;
    }

    clock_gettime(CLOCK_MONOTONIC, &end);

    long duration = (end.tv_sec - start.tv_sec) * 1000000000 +
                    (end.tv_nsec - start.tv_nsec);

    // 如果执行时间异常, 可能在单步调试
    if (duration > NORMAL_EXECUTION_TIME * 10) {
        single_step_detected();
    }
}

void start_timing_monitor() {
    std::thread([]() {
        while (true) {
            std::this_thread::sleep_for(std::chrono::seconds(5));

            // 定期检测系统调用时间
            struct timespec start, end;
            clock_gettime(CLOCK_MONOTONIC, &start);
            getpid(); // 简单系统调用
            clock_gettime(CLOCK_MONOTONIC, &end);

            long syscall_time = (end.tv_sec - start.tv_sec) * 1000000000 +
                               (end.tv_nsec - start.tv_nsec);

            if (syscall_time > NORMAL_SYSCALL_TIME * 5) {
                // 系统调用被拦截或调试
                syscall_hooking_detected();
            }
        }
    }).detach();
}
```

3.4 反调试响应策略

```
enum class AntiDebugResponse {  
    EXIT_SILENTLY,  
    CORRUPT_DATA,  
    FAKE_EXECUTION,  
    CRASH_GRACEFULLY,  
    REPORT_TO_SERVER  
};  
  
void execute_anti_debug_response() {  
    static int detection_count = 0;  
    detection_count++;  
  
    // 根据检测次数选择不同响应策略  
    AntiDebugResponse response = select_response_strategy(detection_count);  
  
    switch (response) {  
        case AntiDebugResponse::EXIT_SILENTLY:  
            _exit(0);  
            break;  
  
        case AntiDebugResponse::CORRUPT_DATA:  
            corrupt_critical_data();  
            break;  
  
        case AntiDebugResponse::FAKE_EXECUTION:  
            enter_fake_execution_mode();  
            break;  
  
        case AntiDebugResponse::CRASH_GRACEFULLY:  
            trigger_controlled_crash();  
            break;  
  
        case AntiDebugResponse::REPORT_TO_SERVER:  
            report_debug_attempt();  
            _exit(1);  
            break;  
    }  
}  
  
void corrupt_critical_data() {  
    // 破坏关键数据结构, 使分析结果无效  
    extern char critical_data_start[];  
    extern char critical_data_end[];  
  
    size_t size = critical_data_end - critical_data_start;  
    for (size_t i = 0; i < size; i++) {  
        critical_data_start[i] ^= 0xFF;  
    }  
}  
  
void enter_fake_execution_mode() {  
    // 进入虚假执行模式, 返回错误分析结果
```

```
global_fake_mode = true;  
  
// 修改函数指针, 指向虚假实现  
redirect_function_calls();  
}
```

4. 分析与绕过方法

4.1 静态分析 init_array

```
# 分析 init_array 段的工具
import subprocess
from elftools.elf.elffile import ELFFile

class InitArrayAnalyzer:
    def __init__(self, so_path):
        self.so_path = so_path
        self.init_functions = []

    def analyze_init_array():
        # 使用 readelf 获取 init_array 信息
        result = subprocess.run(['readelf', '-d', self.so_path],
                               capture_output=True, text=True)

        for line in result.stdout.split('\n'):
            if 'INIT_ARRAY' in line:
                # 解析 init_array 地址和大小
                self.parse_init_array_info(line)

    def extract_function_addresses(self):
        with open(self.so_path, 'rb') as f:
            elf = ELFFile(f)

            # 找到 .init_array 段
            init_array_section = elf.get_section_by_name('.init_array')
            if init_array_section:
                data = init_array_section.data()

                # 解析函数指针 (8 字节对齐)
                for i in range(0, len(data), 8):
                    if i + 8 <= len(data):
                        func_addr = int.from_bytes(data[i:i+8], 'little')
                        self.init_functions.append(func_addr)
                        print(f"[+] Init function at: 0x{func_addr:x}")

    def disassemble_functions(self):
        # 使用 objdump 反汇编每个初始化函数
        for addr in self.init_functions:
            print(f"\n[+] Disassembling function at 0x{addr:x}")
            subprocess.run(['objdump', '-d', '--start-address', hex(addr),
                           '--stop-address', hex(addr + 0x100), self.so_path])

# 使用示例
analyzer = InitArrayAnalyzer('target.so')
analyzer.analyze_init_array()
analyzer.extract_function_addresses()
analyzer.disassemble_functions()
```

4.2 Frida Hook init_array

```
// Hook init_array 执行
function monitor_init_array() {
    // Hook constructor 函数调用
    var call_constructors = Module.findExportByName(
        "linker64", "_ZN6soinfo17call_constructorsEv"
    );
    if (call_constructors) {
        Interceptor.attach(call_constructors, {
            onEnter: function(args) {
                var soinfo = args[0];
                var soname = get_soname(soinfo);
                console.log("[+] Calling constructors for: " + soname);

                this.soname = soname;
                this.start_time = Date.now();
            },
            onLeave: function(retval) {
                var duration = Date.now() - this.start_time;
                console.log("[+] Constructors completed for " + this.soname +
                    " in " + duration + "ms");
            }
        });
    }
}

// Hook 目标SO 的每个 init_array 函数
var target_module = Process.findModuleByName("libtarget.so");
if (target_module) {
    analyze_init_array_section(target_module);
}
}

function analyze_init_array_section(module) {
    // 解析 ELF 文件找到 init_array 段
    var elf_base = module.base;

    // 获取 Program 头表偏移
    var phoff = elf_base.add(0x20).readU64();
    var phnum = elf_base.add(0x38).readU16();

    // 遍历 Program 头, 查找 PT_DYNAMIC
    for (var i = 0; i < phnum; i++) {
        var ph_addr = elf_base.add(phoff).add(i * 56);
        var p_type = ph_addr.readU32();

        if (p_type === 2) { // PT_DYNAMIC
            var p_vaddr = ph_addr.add(16).readU64();
            var dynamic_addr = elf_base.add(p_vaddr);

            parse_dynamic_section(dynamic_addr, module);
            break;
        }
    }
}
```

```
}

function parse_dynamic_section(dynamic_addr, module) {
    var addr = dynamic_addr;

    while (true) {
        var tag = addr.readU64();
        var val = addr.add(8).readU64();

        if (tag === 0) break; // DT_NULL

        if (tag === 25) { // DT_INIT_ARRAY
            var init_array_addr = module.base.add(val);
            console.log("[+] Found init_array at: " + init_array_addr);

            // Hook init_array 中每个函数
            hook_init_array_functions(init_array_addr, module);
        } else if (tag === 27) { // DT_INIT_ARRAYSZ
            var array_size = val;
            console.log("[+] Init_array size: " + array_size);
        }
    }

    addr = addr.add(16);
}
}

function hook_init_array_functions(init_array_addr, module) {
    var num_functions = 10; // 假设最多 10 个函数

    for (var i = 0; i < num_functions; i++) {
        var func_ptr_addr = init_array_addr.add(i * 8);
        var func_addr = func_ptr_addr.readPointer();

        if (func_addr.isNull()) break;

        console.log("[+] Hooking init function " + i + " at: " + func_addr);

        Interceptor.attach(func_addr, {
            onEnter: function(args) {
                console.log("![] Init function called");

                // 打印调用栈
                console.log(Thread.backtrace(this.context, Backtracer.ACCURATE)
                    .map(DebugSymbol.fromAddress).join('\n'));
            },
            onLeave: function(retval) {
                console.log("![] Init function completed");
            }
        });
    }
}
```

4.3 绕过反调试

```
// 绕过 ptrace 检测
function bypass_ptrace() {
    var ptrace = Module.findExportByName("libc.so", "ptrace");
    if (ptrace) {
        Interceptor.attach(ptrace, {
            onEnter: function(args) {
                var request = args[0].toInt32();
                if (request === 0) { // PTRACE_TRACE
                    console.log("[+] Blocking PTRACE_TRACE");
                    args[0] = ptr(-1);
                }
            },
            onLeave: function(retval) {
                // 始终返回成功
                retval.replace(ptr(0));
            }
        });
    }
}

// 绕过文件检测
function bypass_file_detection() {
    var access = Module.findExportByName("libc.so", "access");
    var openat = Module.findExportByName("libc.so", "openat");

    var blocked_paths = [
        "/data/local/tmp/frida-server",
        "/proc/self/maps",
        "/proc/self/status"
    ];

    if (access) {
        Interceptor.attach(access, {
            onEnter: function(args) {
                var path = args[0].readCString();
                if (blocked_paths.some(p => path.includes(p))) {
                    console.log("[+] Blocking access to: " + path);
                    args[0] = Memory.allocUtf8String("/dev/null");
                }
            }
        });
    }

    if (openat) {
        Interceptor.attach(openat, {
            onEnter: function(args) {
                var path = args[1].readCString();
                if (blocked_paths.some(p => path.includes(p))) {
                    console.log("[+] Blocking openat for: " + path);
                    args[1] = Memory.allocUtf8String("/dev/null");
                }
            }
        });
    }
}
```

```
        });
    }
}

// 绕过时间检测
function bypass_timing_detection() {
    var clock_gettime = Module.findExportByName("libc.so", "clock_gettime");
    if (clock_gettime) {
        var fake_time = {
            sec: 1640995200, // 固定时间戳
            nsec: 0
        };

        Interceptor.attach(clock_gettime, {
            onLeave: function(retval) {
                var timespec = this.context.x1; // 第二个参数
                if (!timespec.isNull()) {
                    // 写入固定时间值
                    timespec.writeU64(fake_time.sec);
                    timespec.add(8).writeU64(fake_time.nsec);

                    // 每次调用略微增加纳秒
                    fake_time.nsec += 1000;
                }
            }
        });
    }
}
```

4.4 Hook 字符串解密

```
// 字符串检测混淆检测
function detect_string_obfuscation(so_path) {
    var module = Process.findModuleByName(so_path);
    if (!module) return;

    // 扫描解密函数模式
    var pattern = "48 89 ?? 48 89 ?? 48 83 ?? ?? 8B ?? ??"; // x64 解密函数模式

    Memory.scan(module.base, module.size, pattern, {
        onMatch: function(address, size) {
            console.log("[+] Found potential decryption function at: " + address);

            Interceptor.attach(address, {
                onEnter: function(args) {
                    console.log("[+] Decryption function called");
                    this.args = Array.prototype.slice.call(args);
                },
                onLeave: function(retval) {
                    // 尝试读取解密结果
                    try {
                        var result = retval.readCString();
                        if (result && result.length > 0 && result.length < 256) {
                            console.log("[+] Decrypted string: " + result);
                        }
                    } catch (e) {
                        // 可能不是字符串
                    }
                }
            });
        },
        onComplete: function() {
            console.log("[+] Decryption function scan completed");
        }
    });
}

// 智能反调试绕过
function intelligent_anti_debug_bypass() {
    // 1. 自动检测并绕过常见反调试技术
    bypass_ptrace();
    bypass_file_detection();
    bypass_timing_detection();

    // 2. 监控 init_array 执行
    monitor_init_array();

    // 3. Hook 字符串解密
    detect_string_obfuscation("libtarget.so");

    // 4. 设置定期检查, 处理新反调试机制
    setInterval(function() {
        check_new_anti_debug_mechanisms();
    }, 1000);
}
```

```
    }, 5000);
}

function check_new_anti_debug_mechanisms() {
    // 检测新反调试线程
    var threads = Process.enumerateThreads();
    threads.forEach(function(thread) {
        // 检查线程调用栈是否包含反调试函数
        var backtrace = Thread.backtrace(thread.context, Backtracer.ACCURATE);
        // 分析并处理...
    });
}
```

5. 高级防护策略

5.1 多层级保护机制

```
// 实现多层次保护机制
class ComprehensiveProtection {
private:
    static bool stage1_passed;
    static bool stage2_passed;
    static bool stage3_passed;

public:
    // 第一阶段: 基础检测
    __attribute__((constructor(101)))
    static void protection_stage1() {
        if (basic_anti_debug_check()) {
            stage1_passed = true;
            decrypt_stage2_key();
        } else {
            enter_decoy_mode();
        }
    }

    // 第二阶段: 深度检测
    __attribute__((constructor(102)))
    static void protection_stage2() {
        if (!stage1_passed) return;

        if (advanced_detection()) {
            stage2_passed = true;
            unlock_critical_functions();
        } else {
            corrupt_stage2_data();
        }
    }

    // 第三阶段: 运行时保护
    __attribute__((constructor(103)))
    static void protection_stage3() {
        if (!stage2_passed) return;

        start_runtime_protection();
        stage3_passed = true;
    }

    // 关键函数只有在所有检测通过后才能正常执行
    static bool is_protection_active() {
        return stage1_passed && stage2_passed && stage3_passed;
    }
};
```

5.2 自适应响应系统

```
class AdaptiveResponseSystem {  
    private:  
        enum ThreatLevel {  
            NO_THREAT = 0,  
            LOW_THREAT = 1,  
            MEDIUM_THREAT = 2,  
            HIGH_THREAT = 3,  
            CRITICAL_THREAT = 4  
        };  
  
        static ThreatLevel assess_threat_level() {  
            int threat_score = 0;  
  
            // 各种检测权重评分  
            if (detect_frida()) threat_score += 30;  
            if (detect_debugger()) threat_score += 25;  
            if (detect_hook()) threat_score += 20;  
            if (detect_emulator()) threat_score += 15;  
            if (detect_root()) threat_score += 10;  
  
            if (threat_score >= 80) return CRITICAL_THREAT;  
            if (threat_score >= 60) return HIGH_THREAT;  
            if (threat_score >= 40) return MEDIUM_THREAT;  
            if (threat_score >= 20) return LOW_THREAT;  
            return NO_THREAT;  
        }  
  
    public:  
        static void adaptive_response() {  
            ThreatLevel level = assess_threat_level();  
  
            switch (level) {  
                case CRITICAL_THREAT:  
                    immediate_termination();  
                    break;  
                case HIGH_THREAT:  
                    data_corruption_and_exit();  
                    break;  
                case MEDIUM_THREAT:  
                    fake_execution_mode();  
                    break;  
                case LOW_THREAT:  
                    increased_monitoring();  
                    break;  
                case NO_THREAT:  
                    normal_execution();  
                    break;  
            }  
        }  
};
```

总结

防护方的核心策略：

1. 多层级检测机制，分阶段验证
2. 字符串动态解密，避免静态分析
3. 时间和行为检测，识别调试环境
4. 自适应响应策略，根据威胁等级调整

分析方的应对策略：

1. 静态分析结合动态 Hook
2. 全面的 API 拦截和重定向
3. 时间和环境模拟
4. 自动化绕过脚本开发

这一技术对抗将持续演进，双方都需要不断提升技术水平以应对新的挑战。

[A06] SO 运行时仿真

A06: SO 运行时仿真技术

在高级 Android 逆向工程中，我们经常需要自动化地调用 SO 文件中的加密、签名或校验函数。然而，在真实的设备上通过 Frida Hook 来做这件事，不仅效率低下，而且容易受到反调试和环境检测的阻碍。

SO 运行时仿真（有时被称为“符号执行”的工程化应用）是一种革命性的技术，它通过在 PC 上创建一个模拟的 Android Native 运行环境，直接加载并执行 SO 文件，从而摆脱对真实设备的依赖。

目录

1. 核心架构
2. 推荐项目：unidbg
3. Python + Unicorn 手动实现
4. 基于 chroot 与 linker 的高级仿真
5. 方案对比与选型
6. 总结

核心架构

一个典型的 SO 仿真框架主要由以下几个部分构成：

1. ELF 加载器 (ELF Loader)

这是仿真的基础。它负责像 Android 的 `linker` 一样工作：

功能	说明
解析 ELF	读取 SO 文件的头部、程序头、段表等信息
内存映射	根据程序头 (PT_LOAD) 将 SO 的代码段 (.text) 和数据段 (.data , .bss) 加载到模拟的内存空间中
处理重定位	解析重定位表 (.rel.dyn , .rela.dyn) , 修正所有对内部地址和外部符号的引用

2. CPU 模拟器 (CPU Emulator)

- Unicorn Engine: 这是目前最主流的选择。Unicorn 是一个基于 QEMU 的轻量级、多平台的 CPU 模拟器库
- 指令级控制: Unicorn 允许我们精细地控制执行流程，包括设置寄存器、读写内存、以及通过 Hook 机制在执行到特定指令或地址时触发回调

3. 系统库与环境模拟 (Library & Environment Mocking)

SO 文件不会独立存在，它总是会调用外部函数。仿真框架必须能够"假装"自己是 Android 系统：

Mock 目标	说明
libc.so	提供 malloc , free , memcpy , strlen , printf 等标准 C 库函数
Android 框架库	提供 liblog.so 、 libz.so 、 libcrypto.so 等常用系统库函数
JNI 环境	模拟 JNIEnv 指针和相关函数表 (NewStringUTF , GetFieldID 等)

推荐项目：unidbg

unidbg 是一个非常强大和成熟的、专门用于 Android SO 仿真和符号执行的 Java 开源项目。

unidbg 的优点

特性	说明
高度自动化	内置了完善的 ELF 加载器和常用系统库的 Mock 实现
易于使用	提供了简洁的 API，用户只需几行代码就可以加载 SO、调用函数
JNI 模拟	拥有强大的 JNI 模拟能力，甚至可以调用和 Mock Java 对象的方法
调试与跟踪	支持与 GDB 连接进行远程调试，也可以通过 Hook 机制打印详细的执行日志

unidbg 使用范例

```
// 使用 unidbg 调用 S0 中的签名函数
public class SignatureCalculator {
    public static void main(String[] args) {
        // 1. 创建 Android ARM64 模拟器实例
        Emulator<?> emulator = AndroidEmulatorBuilder.for64Bit().build();
        Memory memory = emulator.getMemory();

        // 2. 加载目标 S0 文件及其依赖
        // unidbg 会自动处理重定位和依赖加载
        Module module = emulator.loadLibrary(new File("libnative-lib.so"));

        // 3. 准备输入数据
        String input = "this is my data to sign";
        // 将输入字符串写入模拟器内存
        Pointer inputPtr = memory.allocateString(input);

        // 4. 调用目标函数
        // callFunction() 会自动处理寄存器和栈设置
        Number result = module.callFunction(emulator, 0x1234, inputPtr,
        input.length());

        // 5. 从模拟器内存中读取结果
        Pointer resultPtr = Pointer.pointer(emulator, result.intValue());
        String signature = resultPtr.getString(0);

        System.out.println("Input: " + input);
        System.out.println("Signature: " + signature);

        // 6. 关闭模拟器
        emulator.close();
    }
}
```

Python + Unicorn 手动实现

如果需要更细粒度的控制，可以使用 Python + Unicorn 从头实现仿真。

依赖库

- Unicorn Engine: CPU 模拟引擎

- pyelftools: 用于解析 ELF 文件
- Python: 胶水语言, 用于编写加载器和 Mock 函数

实现步骤

a. 初始化 Unicorn 环境

```
from unicorn import *
from unicorn.arm64_const import *

# 初始化 ARM64 模拟器
mu = Uc(UC_ARCH_ARM64, UC_MODE_ARM)

# 定义内存区域
BASE_ADDRESS = 0x40000000
STACK_ADDRESS = 0x70000000
STACK_SIZE = 1024 * 1024 # 1MB 栈空间

# 映射内存
mu.mem_map(BASE_ADDRESS, 2 * 1024 * 1024) # 2MB 用于 SO
mu.mem_map(STACK_ADDRESS, STACK_SIZE)

# 设置栈指针 (SP)
mu.reg_write(UC_ARM64_REG_SP, STACK_ADDRESS + STACK_SIZE)
```

b. 加载 ELF 文件

```
from elftools.elf.elffile import ELFFile

def load_so(mu, so_path):
    """加载 SO 文件到模拟器内存"""
    with open(so_path, 'rb') as f:
        elffile = ELFFile(f)
        for segment in elffile.iter_segments():
            if segment.header.p_type == 'PT_LOAD':
                vaddr = segment.header.p_vaddr
                mem_size = segment.header.p_memsz
                file_size = segment.header.p_filesz
                data = segment.data()

                # 将段写入模拟器内存
                mu.mem_write(BASE_ADDRESS + vaddr, data)
                print(f"Loaded segment at {hex(BASE_ADDRESS + vaddr)} size {hex(mem_size)}")

    return BASE_ADDRESS

# 加载 SO
base = load_so(mu, 'libnative-lib.so')
```

c. 实现函数 Mock

```
# 模拟外部函数地址
MOCK_PUTS_ADDR = 0xFFFFFFFFF00001000

# 记录被 Hook 的指令
hooked_instructions = set()

def hook_code(mu, address, size, user_data):
    """代码执行 Hook 回调"""
    if address in hooked_instructions:
        return

    instruction = mu.mem_read(address, size)

    # 简化 BL 指令检查 (ARM64)
    if len(instruction) >= 4 and instruction[3] == 0x94:
        # 计算跳转目标地址 (需要完整解码指令)
        target_addr = calculate_branch_target(address, instruction)

        if target_addr == MOCK_PUTS_ADDR:
            # 1. 读取参数 (ARM64 第一个参数在 X0 寄存器)
            str_ptr = mu.reg_read(UC_ARM64_REG_X0)
            # 2. 从模拟器内存中读取字符串
            str_val = mu.mem_read(str_ptr, 256).split(b'\x00')[0]
            # 3. 执行Mock 功能
            print(f"[+] puts called with: '{str_val.decode()}'")
            # 4. 模拟函数返回
            mu.reg_write(UC_ARM64_REG_PC, mu.reg_read(UC_ARM64_REG_LR))
        else:
            print(f"Warning: Unhandled call to {hex(target_addr)}")

    hooked_instructions.add(address)

# 在 SO 加载区域设置 Hook
mu.hook_add(UC_HOOK_CODE, hook_code, begin=BASE_ADDRESS, end=BASE_ADDRESS + 0x100000)
```

d. 文件系统 Mock

SO 文件可能会访问文件系统。通过 Mock `fopen`、`fread` 等函数，可以完全控制文件访问：

```
class FileSystemMock:  
    """文件系统模拟"""  
    def __init__(self, rootfs_path):  
        self.rootfs = rootfs_path  
        self.file_handles = {}  
        self.next_fd = 100  
  
    def mock_fopen(self, mu, path_ptr, mode_ptr):  
        """模拟 fopen"""  
        # 读取路径参数  
        path = mu.mem_read(path_ptr, 256).split(b'\x00')[0].decode()  
        mode = mu.mem_read(mode_ptr, 8).split(b'\x00')[0].decode()  
  
        # 映射到本地 rootfs  
        local_path = os.path.join(self.rootfs, path.lstrip('/'))  
  
        try:  
            handle = open(local_path, mode)  
            fd = self.next_fd  
            self.file_handles[fd] = handle  
            self.next_fd += 1  
            print(f"[+] fopen('{path}', '{mode}') = {fd}")  
            return fd  
        except FileNotFoundError:  
            print(f"[-] fopen('{path}') failed: file not found")  
            return 0  
  
    def mock_fread(self, mu, buf_ptr, size, count, fd):  
        """模拟 fread"""  
        if fd in self.file_handles:  
            data = self.file_handles[fd].read(size * count)  
            mu.mem_write(buf_ptr, data)  
            return len(data)  
        return 0
```

基于 chroot 与 linker 的高级仿真

这是最接近“真实”的仿真方式。直接利用从 Android 系统中提取出的 `linker64` 程序，在受控的 `chroot` 环境中加载目标 SO。

核心思路

1. 构建 Android Rootfs: 在 Linux 主机上创建一个最小化的 Android 文件系统

2. 编写加载器 (Loader): 用 C 编写一个程序, `chroot` 到 Rootfs 中, 然后启动 `linker64`

3. 编写测试桩 (Test Harness): 编写一个程序, 加载目标 SO 并调用指定函数

实现步骤

a. 准备 Android Rootfs

```
# 在 PC 上创建目录结构
mkdir -p ~/android_rootfs/system/lib64
mkdir -p ~/android_rootfs/system/bin
mkdir -p ~/android_rootfs/data/local/tmp

# 从设备上拉取文件 (以 arm64 为例)
adb pull /system/bin/linker64 ~/android_rootfs/system/bin/
adb pull /system/lib64/libc.so ~/android_rootfs/system/lib64/
adb pull /system/lib64/libdl.so ~/android_rootfs/system/lib64/
adb pull /system/lib64/libm.so ~/android_rootfs/system/lib64/

# 将目标 SO 和测试桩程序放入
cp your_target.so ~/android_rootfs/data/local/tmp/
cp your_harness ~/android_rootfs/data/local/tmp/
```

b. 编写测试桩程序 (harness.c)

```
#include <stdio.h>
#include <dlfcn.h>

// 假设目标 SO 导出函数: char* process_data(const char* input);
typedef const char* (*process_data_func)(const char*);

int main(int argc, char *argv[]) {
    // 在 chroot 环境中, 路径是相对于新根目录的
    void* handle = dlopen("/data/local/tmp/your_target.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "Cannot open library: %s\n", dlerror());
        return -1;
    }

    // 获取函数指针
    process_data_func func = (process_data_func)dlsym(handle, "process_data");
    if (!func) {
        fprintf(stderr, "Cannot find symbol: %s\n", dlerror());
        dlclose(handle);
        return -1;
    }

    // 调用函数并打印结果
    const char* input = "hello from harness";
    const char* result = func(input);
    printf("Result from SO: %s\n", result);

    dlclose(handle);
    return 0;
}
```

c. 编写加载器程序 (loader.c)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    const char* root_dir = "/home/user/android_rootfs"; // 修改为你的 rootfs 路径

    if (chroot(root_dir) != 0) {
        perror("chroot failed");
        return 1;
    }

    // 进入 chroot 后, '/' 就是之前的 root_dir
    chdir("/");

    // 准备 execve 参数
    char *new_argv[] = {
        "/data/local/tmp/harness", // 要执行的程序
        NULL
    };
    char *new_envp[] = {
        "LD_LIBRARY_PATH=/system/lib64", // 告诉 linker 在哪里找 .so
        NULL
    };

    // 使用 linker64 来执行测试桩程序
    execve("/system/bin/linker64", new_argv, new_envp);

    // 如果 execve 成功, 这行代码永远不会被执行
    perror("execve failed");
    return 1;
}
```

注意事项

- 权限要求: `chroot` 操作需要 root 权限
- 通信机制: 主机与被仿真进程的通信需要借助文件、管道或 Socket 等 IPC 机制
- 架构匹配: 需要在相同架构的 Linux 主机上运行 (如 ARM64 主机运行 ARM64 SO)

方案对比与选型

方案	优点	复杂度	适用场景
unidbg (Java)	成熟稳定, JNI 模拟完善, API 简洁, 社区活跃	低	快速分析、JNI 函数调用、通用 Android SO 仿真
Python + Unicorn	灵活可定制, 完全控制执行流程, 跨平台, 帮助理解原理	中	纯 Native 算法逆向、安全研究、Fuzzing、学习目的
C + chroot + Harness	保真度最高, 性能最好, 直接利用系统原生 linker	高	环境要求苛刻的 SO、需要 TLS 初始化、追求极致性能

总结

SO 运行时仿真是一项高级但回报巨大的技术。它将逆向分析从繁琐的手工调试和 Hook 中解放出来，带入了自动化、可大规模扩展的新阶段。

关键要点：

1. 核心组件: ELF 加载器 + CPU 模拟器 + 系统库 Mock
2. 推荐工具: unidbg 是最成熟的 Android SO 仿真框架
3. 进阶方案: Python + Unicorn 提供更细粒度控制, chroot 方案提供最高保真度
4. 应用场景: 加密算法分析、签名函数调用、自动化测试、Fuzzing

对于需要频繁调用 Native 函数、分析复杂算法的场景，掌握 SO 仿真技术是必不可少的技能。

工程实践

[E01] 框架与中间件

E01: 框架、工具与中间件

在复杂的逆向工程和数据采集中，单纯依靠基础工具往往效率低下。为了处理大规模的任务、管理复杂的依赖和保证流程的稳定性，我们需要引入“工程化”的思维，利用成熟的框架和中间件来构建健壮、可扩展的分析系统。

本节内容将聚焦于那些能将单个脚本提升为工业级解决方案的关键技术，例如：

- 消息队列 (Message Queues): 如 RabbitMQ，用于解耦任务的生产者和消费者，实现异步处理和削峰填谷。
- 数据存储 (Data Storage): 如 MongoDB 或 PostgreSQL，用于结构化地存储分析结果，方便后续的查询和二次开发。
- 缓存系统 (Caching Systems): 如 Redis，用于缓存常用数据，加速热点路径的访问。
- 爬虫框架 (Crawling Frameworks): 如 Scrapy，提供了一整套用于网络数据提取的架构，包括请求调度、中间件处理和数据管道。

通过组合这些工具，我们可以搭建起一个能够处理海量设备、执行复杂任务并高效存储结果的强大平台。

[E02] 消息队列

E02: 消息队列 (Message Queue)

消息队列 (MQ) 是大型分布式系统中用于服务间异步通信的核心组件。在规模化的逆向分析和数据采集中，它扮演着“缓冲池”和“解耦器”的关键角色，确保数据流的稳定、高效和可靠。

1. 核心概念与作用

a) 为什么需要消息队列？

想象一个场景：你有 100 台爬虫节点（生产者）在高速抓取数据，同时有 10 个数据处理节点（消费者）负责清洗和入库。如果让生产者直接调用消费者的 API，会产生几个问题：

- 性能耦合：消费者的处理速度会直接限制生产者的抓取速度。如果数据库写入缓慢，整个爬虫集群都得等。
- 峰值压力：如果短时间内抓取到大量数据（流量洪峰），可能会瞬间压垮消费者服务。
- 服务依赖：如果消费者服务宕机，所有生产者都会失败，数据会丢失。

b) 消息队列的解决方案

MQ 在生产者和消费者之间增加了一个中间层，解决了以上所有问题：

- 异步解耦：生产者只需将消息（如“一个待处理的数据包”）扔进队列即可，无需关心谁在消费、何时消费。
- 削峰填谷：流量洪峰到来时，消息会先在队列中积压。消费者可以按照自己的节奏平稳地进行处理，避免了系统崩溃。

- 可靠性与冗余: 即使消费者宕机, 消息仍然安全地存储在队列中。当消费者恢复后, 可以继续处理, 保证了数据不丢失。

2. 主流消息队列方案

a) Kafka

- 定位: 一个分布式的、分区的、多副本的、基于 Zookeeper 的日志提交系统 (Commit Log)。
- 核心特点:
 - 极致的吞吐量: 设计目标就是为了处理海量日志数据, 拥有无与伦比的写入和读取性能, 是大数据领域的首选。
 - 发布-订阅模型: 消息以"主题 (Topic)"进行分类。生产者向一个 Topic 发送消息, 多个消费者组 (Consumer Group) 可以独立地订阅和消费同一个 Topic 的消息, 互不干扰。
 - 持久化与回溯: 消息在 Kafka 中是持久化存储的。消费者可以根据需要"回溯"到任意时间点 (Offset) 重新消费数据, 这对于数据重处理和故障恢复非常有用。
- 适用场景:
 - 需要处理海量数据流的日志收集 (Log Ingestion)。
 - 作为 Spark Streaming 或 Flink 等实时计算框架的数据源。
 - 构建大规模数据管道的总线。

b) RabbitMQ

- 定位: 一个实现了 AMQP (高级消息队列协议) 的、功能丰富的消息代理 (Message Broker)。
- 核心特点:
 - 灵活的路由: 拥有强大的交换机 (Exchange) 和路由键 (Routing Key) 机制, 可以实现非常复杂的路由逻辑 (如 fanout, direct, topic, headers)。

- 功能全面: 支持消息确认、优先级队列、延迟队列、死信队列等企业级特性。
- 可靠性: 提供了强大的消息确认机制, 能确保消息"至少被成功消费一次"。
- 适用场景:
 - 业务逻辑复杂, 需要精细化控制消息路由的场景。
 - 对消息投递的可靠性要求极高的金融或事务性系统。
 - 需要使用延迟队列等高级特性的业务。

c) Redis

- 定位: 一个高性能的内存数据库, 但其 `List` 和 `Pub/Sub` 功能使其可以作为一个轻量级的消息队列使用。
- 核心特点:
 - 简单快速: 配置简单, 读写性能极高 (基于内存)。
 - 功能有限: 不支持复杂路由, 可靠性保证较弱 (如 `Pub/Sub` 不保证消息必达), 消息积压能力受内存限制。
- 适用场景:
 - 系统规模不大, 对可靠性要求不高, 但对实时性要求很高的场景。
 - 作为任务队列 (如 Celery 的 Broker)。
 - 实现简单的实时通知或聊天功能。

总结

在工程化体系中, 选择哪种 MQ 取决于具体的业务需求:

- 追求极致的吞吐量和大数据生态兼容性, 选择 `Kafka`。
- 追求灵活的路由和业务功能的丰富性, 选择 `RabbitMQ`。
- 追求简单、轻量和极致的低延迟, `Redis` 是一个不错的备选项。

[E03] Redis 常用命令

E03: Redis 常用命令备忘录

Redis 是一个开源的、基于内存的、高性能的键值存储系统。它支持多种数据结构，如字符串、哈希、列表、集合和有序集合。本备忘录旨在提供常用命令的快速参考。

目录

1. 连接与服务器管理
2. 键 (Key) 操作
3. 字符串 (String)
4. 哈希 (Hash)
5. 列表 (List)
6. 集合 (Set)
7. 有序集合 (Sorted Set / ZSet)
8. Redis 架构演进
9. Redis 集群详解
10. 布隆过滤器
11. Stream 流数据结构
12. 面试高频考点

连接与服务器管理

命令	描述
<code>redis-cli</code>	启动 Redis 命令行客户端
<code>redis-cli -h <host> -p <port> -a <password></code>	连接到指定的 Redis 实例
<code>PING</code>	测试服务器是否仍在运行，返回 <code>PONG</code>
<code>AUTH <password></code>	验证连接密码
<code>SELECT <index></code>	选择数据库 (默认 0-15)
<code>FLUSHDB</code>	清空当前数据库的所有键
<code>FLUSHALL</code>	清空所有数据库的所有键
<code>INFO</code>	获取服务器的信息和统计数据

键 (Key) 操作

命令	描述
<code>KEYS <pattern></code>	查找所有符合给定模式的键 (如 <code>KEYS *</code> , <code>KEYS user:*</code>) (慎用, 会阻塞)
<code>SCAN <cursor> [MATCH pattern] [COUNT count]</code>	迭代数据库中的键, 比 <code>KEYS</code> 更安全
<code>EXISTS <key></code>	检查给定键是否存在
<code>DEL <key> [key ...]</code>	删除一个或多个键
<code>TYPE <key></code>	返回键所存储的值的类型 (string, hash, list, set, zset)
<code>TTL <key></code>	以秒为单位, 返回给定键的剩余生存时间
<code>EXPIRE <key> <seconds></code>	为给定键设置生存时间
<code>PERSIST <key></code>	移除给定键的生存时间, 使其永久保存
<code>RENAME <key> <newkey></code>	修改键的名称

字符串 (String)

字符串是 Redis 最基本的数据类型, 可以存储任何类型的数据, 如文本、序列化的 JSON 或二进制数据。

命令	描述
SET <key> <value>	设置指定键的值
GET <key>	获取指定键的值
SETEX <key> <seconds> <value>	设置键值对并指定过期时间
SETNX <key> <value>	只有在键不存在时才设置键的值
MSET <key1> <value1> [key2 value2 ...]	同时设置一个或多个键值对
MGET <key1> [key2 ...]	获取所有给定键的值
INCR <key>	将键中储存的数字值增一 (原子操作)
DECR <key>	将键中储存的数字值减一 (原子操作)
INCRBY <key> <increment>	将键所储存的值加上指定的增量值

哈希 (Hash)

哈希是一个键值对的集合，非常适合用于存储对象。

命令	描述
HSET <key> <field> <value>	将哈希表中字段的值设为 value
HGET <key> <field>	获取存储在哈希表中指定字段的值
HMSET <key> <f1> <v1> [f2 v2 ...]	同时将多个 field-value 对设置到哈希表中
HMGET <key> <field1> [field2 ...]	获取所有给定字段的值
HGETALL <key>	获取在哈希表中指定键的所有字段和值
HKEYS <key>	获取哈希表中的所有字段
HVALS <key>	获取哈希表中的所有值
HDEL <key> <field1> [field2 ...]	删除一个或多个哈希表字段
HEXISTS <key> <field>	查看哈希表的指定字段是否存在

列表 (List)

列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

命令	描述
LPUSH <key> <value1> [value2 ...]	将一个或多个值插入到列表头部
RPUSH <key> <value1> [value2 ...]	将一个或多个值插入到列表尾部
LPOP <key>	移出并获取列表的第一个元素
RPOP <key>	移出并获取列表的最后一个元素
LLEN <key>	获取列表的长度
LRANGE <key> <start> <stop>	获取列表指定范围内的元素
LINDEX <key> <index>	通过索引获取列表中的元素
LSET <key> <index> <value>	通过索引设置列表元素的值
LTRIM <key> <start> <stop>	对列表进行修剪，只保留指定区间内的元素

集合 (Set)

集合是字符串类型的无序集合。集合成员是唯一的，不能出现重复的数据。

命令	描述
SADD <key> <member1> [member2 ...]	向集合添加一个或多个成员
SMEMBERS <key>	返回集合中的所有成员
SISMEMBER <key> <member>	判断 member 元素是否是集合的成员
SCARD <key>	获取集合的成员数
SREM <key> <member1> [member2 ...]	移除集合中一个或多个成员
SPOP <key> [count]	随机移除并返回集合中一个或多个成员
SUNION <key1> [key2 ...]	返回所有给定集合的并集
SINTER <key1> [key2 ...]	返回所有给定集合的交集
SDIFF <key1> [key2 ...]	返回所有给定集合的差集

有序集合 (Sorted Set / ZSet)

有序集合和集合一样是字符串类型元素的集合，且不允许重复的成员。不同的是每个元素都会关联一个 `double` 类型的分数 (score)。

命令	描述
ZADD <key> <score1> <member1> [...]	向有序集合添加一个或多个成员
ZRANGE <key> <start> <stop> [WITHSCORES]	通过索引区间返回成员 (分数递增)
ZREVRANGE <key> <start> <stop> [WITHSCORES]	返回指定区间内的成员 (分数递减)
ZRANGEBYSCORE <key> <min> <max>	通过分数返回指定区间内的成员
ZCARD <key>	获取有序集合的成员数
ZSCORE <key> <member>	返回成员的 score 值
ZREM <key> <member1> [member2 ...]	移除一个或多个成员
ZCOUNT <key> <min> <max>	计算指定分数区间的成员数

Redis 架构演进

版本发展

版本	发布时间	主要特性
Redis 1.0	2009 年	基础键值存储, 5 种基本数据结构
Redis 2.0	2010 年	引入虚拟内存、发布订阅
Redis 2.6	2012 年	Lua 脚本支持、过期键处理优化
Redis 2.8	2013 年	部分重同步、Sentinel 高可用
Redis 3.0	2015 年	Redis Cluster 集群支持
Redis 4.0	2017 年	模块系统、内存优化、混合持久化
Redis 5.0	2018 年	Stream 数据结构、动态 HZ
Redis 6.0	2020 年	多线程 I/O、ACL 权限控制、SSL
Redis 7.0	2022 年	Redis Functions、多 ACL 用户

架构演进路径

1. 单机模式 (Single Instance)

最简单的部署方式，适合开发和测试环境。

2. 主从复制 (Master-Slave)

数据异步复制到从节点，提供读写分离能力。

3. Sentinel 高可用 (Redis Sentinel)

监控主从节点，自动进行故障转移。

4. 集群模式 (Redis Cluster)

数据分片存储，支持水平扩展。

脑裂问题 (Split-Brain)

定义：脑裂是指在分布式系统中，由于网络分区或节点故障，导致系统中出现多个“大脑”（多个节点都认为自己是主节点）的情况。

Redis 中的脑裂场景：

网络分区前：

Sentinel1,2,3 ↔ Master ↔ Slave1,2

网络分区后：

PartitionA: Sentinel1 ↔ Master (原主节点)

PartitionB: Sentinel2,3 ↔ Slave1 ↔ Slave2 (选举新主节点)

Redis 脑裂预防机制：

1. Sentinel 奇数部署：确保故障转移时有明确的多数派

```
# 推荐配置：至少 3 个 Sentinel  
Sentinel1, Sentinel2, Sentinel3
```

1. 配置写入限制：

```
# 至少需要 2 个 Slave 节点，最大延迟 10 秒  
min-slaves-to-write 2  
min-slaves-max-lag 10
```

1. 客户端配置：

```
sentinel = Sentinel([
    ('localhost', 26379),
    ('localhost', 26380),
    ('localhost', 26381)
])
master = sentinel.master_for('mymaster', socket_timeout=0.1)
```

Redis 集群详解

Sentinel 模式

核心功能

1. 监控 (Monitoring): 监控 master 和 slave 健康状态
2. 通知 (Notification): 故障时通知管理员
3. 自动故障转移 (Automatic Failover): 自动选举新 master
4. 配置提供 (Configuration Provider): 为客户端提供当前 master 地址

工作原理

```
# Sentinel 配置文件
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 30000
sentinel parallel-syncs mymaster 1
sentinel failover-timeout mymaster 180000
```

故障转移流程:

1. 主观下线 (SDOWN): 单个 Sentinel 认为主节点不可用
2. 客观下线 (ODOWN): 多数 Sentinel 确认主节点不可用
3. 领导选举: Sentinel 之间选举领导者执行故障转移
4. 新主选择: 选择最优 slave 提升为 master
5. 配置更新: 更新所有节点配置

Cluster 集群模式

集群特性

- 去中心化: 无单点故障
- 数据分片: 自动数据分布
- 高可用: master 故障自动切换
- 在线扩缩容: 支持动态添加/删除节点

数据分片算法

```
# 计算 Key 哈希 Slot  
HASH_SLOT = CRC16(key) % 16384  
  
# Slot 分配示例 (3 个Master 节点)  
Master1: 0-5461 (5462 Slot)  
Master2: 5462-10923 (5462 Slot)  
Master3: 10924-16383 (5460 Slot)
```

集群操作命令

```
# 创建集群  
redis-cli --cluster create \  
127.0.0.1:7000 127.0.0.1:7001 127.0.0.1:7002 \  
127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 \  
--cluster-replicas 1  
  
# 查看集群信息  
CLUSTER INFO  
CLUSTER NODES  
  
# 重新分片  
redis-cli --cluster reshard 127.0.0.1:7000  
  
# 添加节点  
redis-cli --cluster add-node 127.0.0.1:7006 127.0.0.1:7000
```

布隆过滤器

布隆过滤器是一种概率性数据结构，用于高效判断一个元素是否在集合中。

核心特性

- 误报率: 可能误判存在, 但不会误判不存在
- 空间效率: 使用位数组, 空间复杂度低
- 时间复杂度: $O(k)$ 查询时间, k 为哈希函数数量

实现原理

```
# 基本流程
# 1. 初始化: 创建 m 位位数组, k 个哈希函数
# 2. 添加元素: 对元素计算 k 个哈希值, 设置对应位为 1
# 3. 查询元素: 计算 k 个哈希值, 检查对应位是否都为 1
```

参数计算

```
# 最优位数组大小
m = -n * ln(p) / (ln(2))^2

# 最优哈希函数数量
k = (m/n) * ln(2)

# 其中: n=元素数量, p=误报率, m=位数组大小, k=哈希函数数量
```

Redis 布隆过滤器

```
# 加载 RedisBloom 模块
MODULE LOAD /path/to/redisbloom.so

# 创建布隆过滤器
BF.RESERVE myfilter 0.01 10000

# 添加元素
BF.ADD myfilter "user123"
BF.MADD myfilter "user1" "user2" "user3"

# 检查元素
BF.EXISTS myfilter "user123"
BF.MEXISTS myfilter "user1" "user2"

# 获取信息
BF.INFO myfilter
```

应用场景

1. 缓存穿透防护: 过滤不存在的请求
2. 爬虫 URL 去重: 避免重复爬取
3. 垃圾邮件过滤: 快速判断发件人
4. 推荐系统: 已推荐内容过滤

Stream 流数据结构

Stream 是 Redis 5.0 引入的新数据结构，主要用于消息队列和事件流处理。

核心特性

- 持久化消息队列: 消息持久化存储
- 消费者组: 支持多消费者协作
- 消息确认: 支持消息确认机制

- 历史消息: 可以查询历史消息

消息 ID 结构

<毫秒时间戳>-<序列号>

例如: 1609459200000-0

基本操作

```
# 添加消息
XADD mystream * field1 value1 field2 value2
XADD mystream 1609459200000-0 user "john" action "login"

# 读取消息
XREAD COUNT 2 STREAMS mystream 0
XREAD BLOCK 1000 STREAMS mystream $ # 阻塞读取新消息

# 查看 Stream 信息
XINFO STREAM mystream
XLEN mystream

# 范围查询
XRANGE mystream - +
XRANGE mystream 1609459200000 1609459300000
```

消费者组操作

```
# 创建消费者组
XGROUP CREATE mystream mygroup $ MKSTREAM

# 消费者读取
XREADGROUP GROUP mygroup consumer1 COUNT 1 STREAMS mystream >

# 确认消息
XACK mystream mygroup 1609459200000-0

# 查看消费者组信息
XINFO GROUPS mystream
XINFO CONSUMERS mystream mygroup

# 处理 pending 消息
XPENDING mystream mygroup
XCLAIM mystream mygroup consumer2 1800000 1609459200000-0
```

Stream 与其他队列对比

特性	List	Pub/Sub	Stream
持久化	支持	不支持	支持
多消费者	不支持	支持	支持
消息确认	不支持	不支持	支持
历史消息	支持	不支持	支持
消费者组	不支持	不支持	支持

面试高频考点

持久化机制

RDB (Redis Database)

```
# 配置文件设置
save 900 1      # 900 秒内至少 1 个 key 变化
save 300 10     # 300 秒内至少 10 个 key 变化
save 60 10000   # 60 秒内至少 10000 个 key 变化

# 手动触发
SAVE      # 同步保存 (阻塞)
BGSAVE   # 异步保存 (后台)
```

AOF (Append Only File)

```
# 配置选项
appendonly yes
appendfsync always    # 每次写入立即同步
appendfsync everysec  # 每秒同步一次 (推荐)
appendfsync no         # 由 OS 决定同步时机
```

混合持久化 (Redis 4.0+)

```
aof-use-rdb-preamble yes
```

内存淘汰策略

```
# 配置最大内存
maxmemory 2gb

# 淘汰策略
maxmemory-policy allkeys-lru
```

策略	说明
<code>noeviction</code>	不淘汰，内存满时返回错误
<code>allkeys-lru</code>	所有 key 中淘汰最近最少使用的
<code>allkeys-lfu</code>	所有 key 中淘汰最少频率的
<code>volatile-lru</code>	有过期时间的 key 中淘汰最近最少使用的
<code>volatile-lfu</code>	有过期时间的 key 中淘汰最少频率的
<code>volatile-random</code>	有过期时间的 key 中随机淘汰
<code>volatile-ttl</code>	淘汰即将过期的 key

缓存问题解决方案

1. 缓存穿透

- 问题: 查询不存在的数据, 绕过缓存直接查数据库
- 解决方案:
 - 布隆过滤器预过滤
 - 空值缓存 (设置较短过期时间)
 - 参数校验

2. 缓存雪崩

- 问题: 大量缓存同时失效, 数据库压力激增
- 解决方案:
 - 随机过期时间
 - 缓存预热
 - 多级缓存
 - 限流降级

3. 缓存击穿

- 问题: 热点数据过期, 大量请求直达数据库
- 解决方案:
 - 互斥锁重建缓存
 - 异步更新缓存
 - 热点数据永不过期

分布式锁实现

基于 SETNX 的简单锁

```
# 加锁
SET lock_key unique_value PX 30000 NX

# 释放锁 (Lua 脚本保证原子性)
if redis.call("get", KEYS[1]) == ARGV[1] then
    return redis.call("del", KEYS[1])
else
    return 0
end
```

Redlock 算法

1. 获取当前时间戳
2. 依次向 N 个 Redis 实例请求加锁
3. 超过 $N/2+1$ 个实例加锁成功才算成功
4. 加锁总时间要小于锁过期时间
5. 释放所有实例上的锁

性能优化

1. 合理的数据结构
 - 小数据量使用 ziplist 编码 (节省内存)

-
- 大数据量使用 hashtable 编码（提高性能）

2. 批量操作

```
# 使用 pipeline 减少网络往返
PIPELINE
SET key1 value1
SET key2 value2
EXEC

# 使用 MGET/MSET 批量操作
MSET key1 value1 key2 value2
MGET key1 key2
```

3. 配置优化

```
# 配置优化
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
list-max-ziplist-size -2
set-max-intset-entries 512
```

数据库技术选型对比

关系型数据库

MySQL

方面	描述
存储引擎	InnoDB (事务)、MyISAM (性能)
事务支持	完整 ACID 支持
复制	主从复制、主主复制
适用场景	Web 应用、电商系统、金融系统

PostgreSQL

方面	描述
数据类型	丰富的内置类型 (JSON、数组、地理)
索引类型	B-tree、Hash、GiST、GIN、BRIN
并发控制	MVCC 多版本并发控制
适用场景	复杂查询、数据分析、地理信息系统

NoSQL 数据库

MongoDB

方面	描述
数据模型	BSON 文档
分片	自动分片 (Auto-Sharding)
复制	副本集 (Replica Set)
适用场景	内容管理、实时分析、物联网

Cassandra

方面	描述
数据模型	宽列存储
一致性	可调一致性
分区	一致性哈希
适用场景	时序数据、日志系统、推荐系统

数据库对比总结

数据库	类型	一致性	扩展性	查询能力	适用场景
MySQL	关系型	强一致	垂直扩展	很强	通用业务
PostgreSQL	关系型	强一致	垂直扩展	很强	复杂分析
Redis	键值	最终一致	水平扩展	弱	缓存、会话
MongoDB	文档	强一致	水平扩展	中等	内容管理
Cassandra	列族	可调一致	线性扩展	弱	高并发写入

选型原则

1. 业务需求优先: 根据具体业务场景选择
2. 团队能力: 考虑团队的技术栈和维护能力
3. 成本控制: 综合考虑开发、运维、硬件成本
4. 未来扩展: 预留技术演进空间

最佳实践

- 读多写少: MySQL/PostgreSQL + Redis
- 写多读少: Cassandra/MongoDB + Redis
- 复杂查询: PostgreSQL + 数据仓库
- 实时分析: HBase + Spark/Flink
- 混合负载: 多数据库架构 + 数据同步

[E04] 风控 SDK 编译指南

E04: 风控 SDK 编译指南

Risk Control SDK 是一个用于移动应用的设备指纹识别和安全评估系统，基于 JNI 架构实现。本指南将详细说明如何编译和构建该 SDK。

目录

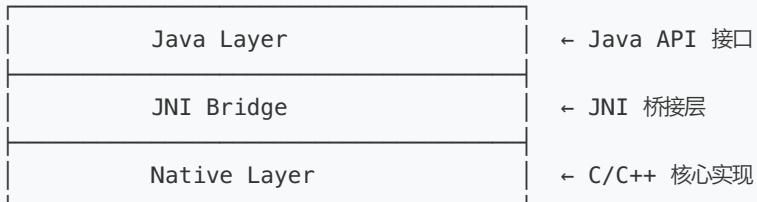
1. 项目概述
2. 环境准备
3. 项目结构
4. 编译步骤
5. 构建选项
6. 多平台编译
7. 问题排查

项目概述

Risk Control SDK 提供以下核心功能：

- 设备指纹识别：硬件、软件、网络、行为指纹采集
- 安全检测：模拟器、Root、调试器、Hook 框架检测
- 反逆向工程：代码混淆、字符串加密、运行时完整性监控
- 风险评估：基于多维度数据的风险评分算法

技术架构



环境准备

基础开发工具

```
# Ubuntu/Debian
sudo apt update
sudo apt install -y build-essential cmake git

# macOS
brew install cmake
xcode-select --install

# Windows (使用MSYS2)
pacman -S base-devel cmake git
```

JDK 安装

```
# Ubuntu/Debian
sudo apt install -y openjdk-8-jdk

# macOS
brew install openjdk@8

# 验证安装
java -version
javac -version
```

Android NDK (可选, 用于 Android 平台)

```
# 下载 NDK  
wget https://dl.google.com/android/repository/android-ndk-r25c-linux.zip  
unzip android-ndk-r25c-linux.zip  
export ANDROID_NDK=/path/to/android-ndk-r25c  
  
# 下载 Android SDK (可选, 用于测试)  
wget https://dl.google.com/android/repository/commandlinetools-  
linux-9477386_latest.zip
```

内存检查工具 (仅 Linux)

```
sudo apt install -y valgrind
```

环境变量配置

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64  
export PATH=$JAVA_HOME/bin:$PATH
```

项目结构

```
risk-control-sdk/
├── CMakeLists.txt          # CMake 构建配置
├── docs/                   # 文档目录
│   └── ArtMethod_Direct_Registration.md
│       └── diagrams/           # 架构图
├── examples/               # 示例代码
│   └── ArtMethodDemo.java
│       └── RiskControlDemo.java
├── include/                # 头文件目录
├── scripts/                # 构建脚本
│   └── build.sh.in          # 通用构建脚本模板
│       └── build-android.sh.in # Android 构建脚本模板
└── src/                    # 源代码
    ├── java/                # Java 层实现
    │   └── com/riskcontrol/
    │       ├── DeviceFingerprint.java
    │       ├── RiskControlSDK.java
    │       ├── RiskScore.java
    │       └── SecurityResult.java
    └── native/               # Native 层实现
        ├── anti_reverse.c/.h  # 反逆向工程保护
        ├── art_method_hook.c/.h # ART 方法 Hook
        ├── risk_control.c/.h   # 主要风控逻辑
        └── svc_syscall.c/.h    # 系统调用处理
```

编译步骤

1. 克隆项目

```
cd /path/to/android_reversing/playground
git clone <repository-url> risk-control-sdk # 或使用现有代码
cd risk-control-sdk
```

2. 配置 CMake

```
mkdir build && cd build

# 基础配置
cmake ..

# 或指定详细选项
cmake -DCMAKE_BUILD_TYPE=Release \
      -DENABLE_DEBUG=OFF \
      -DENABLE_ANTI_REVERSE=ON \
      -DENABLE_SVC_SYSCALLS=ON \
      ..
```

3. 编译

```
# 编译所有目标
make -j$(nproc)

# 或编译特定目标
make riskcontrol      # 编译Native 库
make compile_java     # 编译 Java 代码
make create_jar        # 创建 JAR 包
make example          # 编译示例程序
```

4. 验证输出

```
ls build/

# 应该看到:
# - libriskcontrol.so (Linux) 或 libriskcontrol.jnilib (macOS)
# - RiskControlSDK.jar
# - examples/

# 运行测试
./examples/RiskControlDemo
```

Android 平台编译

验证 NDK 配置

```
echo $ANDROID_NDK
# 应输出: /path/to/android-ndk-r25c

# 验证NDK 工具链
$ANDROID_NDK/ndk-build --version
```

生成构建脚本

```
cmake -P scripts/generate_build_scripts.cmake

# 执行 Android 构建
chmod +x build-android.sh
./build-android.sh
```

手动编译

```
mkdir build-android && cd build-android

# 配置 Android 工具链
cmake -DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK/build/cmake/android.toolchain.cmake \
      -DANDROID_ABI=arm64-v8a \
      -DANDROID_PLATFORM=android-21 \
      -DANDROID_NDK=$ANDROID_NDK \
      -DBUILD_ANDROID=ON \
      -DCMAKE_BUILD_TYPE=Release \
      ..

# 编译
make -j$(nproc)
```

多架构编译

```
ARCHITECTURES=("arm64-v8a" "armeabi-v7a" "x86" "x86_64")

for ARCH in "${ARCHITECTURES[@]}"; do
    echo "Building for $ARCH..."
    mkdir -p build-android-$ARCH
    cd build-android-$ARCH

    cmake -DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK/build/cmake/android.toolchain.cmake \
        -DANDROID_ABI=$ARCH \
        -DANDROID_PLATFORM=android-21 \
        -DANDROID_NDK=$ANDROID_NDK \
        -DBUILD_ANDROID=ON \
        -DCMAKE_BUILD_TYPE=Release \
        ..

    make -j$(nproc)
    cd ..
done
```

构建选项

CMake 选项说明

选项	默认值	说明
CMAKE_BUILD_TYPE	Release	构建类型: Debug/Release/RelWithDebInfo
ENABLE_DEBUG	OFF	启用调试模式和日志输出
ENABLE_ANTI_REVERSE	ON	启用反逆向工程保护
ENABLE_SVC_SYSCALLS	ON	启用 SVC 系统调用支持
BUILD_ANDROID	OFF	Android 平台构建模式
BUILD_EXAMPLES	ON	构建示例程序
BUILD_DOCS	OFF	生成 Doxygen 文档

使用示例

```
# 调试构建
cmake -DCMAKE_BUILD_TYPE=Debug -DENABLE_DEBUG=ON ..

# 最小化构建（禁用保护功能）
cmake -DENABLE_ANTI_REVERSE=OFF -DENABLE_SVC_SYSCALLS=OFF ..

# 发布构建（全功能）
cmake -DCMAKE_BUILD_TYPE=Release \
      -DENABLE_ANTI_REVERSE=ON \
      -DENABLE_SVC_SYSCALLS=ON \
      -DBUILD_EXAMPLES=OFF \
      ..
```

多平台编译

支持的平台

平台	架构	编译器	输出文件
Linux	x86_64	GCC/Clang	libriskcontrol.so
macOS	x86_64/arm64	Clang	libriskcontrol.jnilib
Windows	x86_64	MinGW/MSVC	riskcontrol.dll
Android	arm64-v8a	Android NDK	libriskcontrol.so
Android	armeabi-v7a	Android NDK	libriskcontrol.so
Android	x86/x86_64	Android NDK	libriskcontrol.so

平台特定配置

Linux 编译

```
# Ubuntu/Debian 依赖
sudo apt install -y build-essential cmake openjdk-8-jdk

# 编译
mkdir build-linux && cd build-linux
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j$(nproc)
```

macOS 编译

```
# 编译 (支持 Universal Binary)
mkdir build-macos && cd build-macos
cmake -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE OSX_ARCHITECTURES="x86_64;arm64" \
    ..
make -j$(sysctl -n hw.ncpu)
```

Windows 编译

```
mkdir build-windows && cd build-windows
cmake -G "MinGW Makefiles" -DCMAKE_BUILD_TYPE=Release ..
mingw32-make -j$(nproc)
```

问题排查

问题 1: 找不到 JNI 头文件

错误: fatal error: jni.h: No such file or directory

解决方案:

1. 验证 JDK 安装:

```
java -version
javac -version
```

1. 查找正确 JDK 路径:

```
# macOS Homebrew 安装的 OpenJDK  
ls /usr/local/Cellar/openjdk/  
  
# Ubuntu/Debian  
ls /usr/lib/jvm/  
  
# 手动查找 JNI 头文件  
find /usr -name "jni.h" 2>/dev/null
```

1. 设置正确的 JAVA_HOME:

```
# macOS Homebrew OpenJDK 示例  
export JAVA_HOME=/usr/local/Cellar/openjdk/24.0.1/libexec/openjdk.jdk/Contents/Home  
  
# Ubuntu/Debian 示例  
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```

1. 验证 JNI 头文件存在:

```
ls -la $JAVA_HOME/include/jni.h
```

1. 重新运行 CMake:

```
cmake ..
```

问题 2: system_properties.h 不存在

错误: fatal error: sys/system_properties.h: No such file or directory

解决方案: 添加平台兼容性处理

在 `art_method_hook.c` 中修改:

```
// 原始代码
#include <sys/system_properties.h>

// 修改为平台兼容
#ifndef __ANDROID__
#include <sys/system_properties.h>
#else
// 非 Android 平台替代实现
#define PROP_VALUE_MAX 256
static int __system_property_get(const char* name, char* value) {
    if (strcmp(name, "ro.build.version.sdk") == 0) {
        strcpy(value, "28"); // 模拟 Android API 28
        return strlen(value);
    }
    value[0] = '\0';
    return 0;
}
#endif
```

问题 3: 类型未定义

错误: `unknown type name 'uint32_t'`

解决方案:

```
#include <stdint.h> // 添加标准整数类型支持
```

问题 4: 结构体重复定义

错误: `redefinition of 'native_method_t'`

最佳解决方案: 创建公共类型定义文件

1. 创建 `src/native/common_types.h`:

```
#ifndef COMMON_TYPES_H
#define COMMON_TYPES_H

typedef struct {
    const char* name;
    const char* signature;
    void* fnPtr;
} native_method_t;

#endif
```

1. 在所有需要此类型的头文件中包含:

```
#include "common_types.h"
```

1. 删除其他文件中的重复定义

问题 5: dladdr 函数未声明

错误: `implicit declaration of function 'dladdr'`

解决方案: 启用 GNU 扩展并添加 dl 库链接

1. 在 `art_method_hook.c` 开头添加 (必须在所有 `#include` 之前) :

```
// 启用 GNU 扩展功能以确保 dladdr 可用
#ifndef __GNU_SOURCE
#define __GNU_SOURCE
#endif

#include "art_method_hook.h"
// ... 其他 includes ...
#include <dlfcn.h>
```

1. 在 `CMakeLists.txt` 中添加 dl 库链接:

```
# 链接库  
target_link_libraries(riskcontrol ${JNI_LIBRARIES})  
  
# 在非 Android 平台上链接 dl 库  
if(NOT BUILD_ANDROID)  
    target_link_libraries(riskcontrol dl)  
endif()
```

1. 重新配置项目:

```
rm -f CMakeCache.txt  
cmake ..  
make clean  
make -j8
```

问题 6: 权限问题

```
sudo chown -R $USER:$USER build/
```

编译优化建议

并行编译

```
make -j$(nproc)          # Linux  
make -j$(sysctl -n hw.ncpu)  # macOS
```

使用 Ninja 构建系统

```
# 使用Ninja 构建系统（更快）  
cmake -G Ninja ..  
ninja
```

使用 ccache 加速

```
export CC="ccache gcc"
export CXX="ccache g++"
```

调试技巧

CMake 详细输出

```
cmake --verbose ..
# Make 详细输出
make VERBOSE=1
```

检查库依赖

```
ldd libriskcontrol.so      # Linux
otool -L libriskcontrol.jnilib # macOS

# 检查符号
nm -D libriskcontrol.so
objdump -T libriskcontrol.so
```

Java JNI 调试

```
java -Djava.library.path=./build -verbose:jni RiskControlDemo
```

集成指南

Java 项目集成

```
// 加载 SDK
System.loadLibrary("riskcontrol");

// 使用 SDK
RiskControlSDK sdk = RiskControlSDK.getInstance();
SecurityResult result = sdk.performSecurityCheck();
DeviceFingerprint fingerprint = sdk.getDeviceFingerprint();
RiskScore score = sdk.calculateRiskScore(fingerprint, result);
```

Android 项目集成

在 `build.gradle` 中:

```
android {
    // ...
    sourceSets {
        main {
            jniLibs.srcDirs = ['libs']
        }
    }
}

dependencies {
    implementation files('libs/RiskControlSDK.jar')
}
```

复制编译产物:

```
cp build/RiskControlSDK.jar /target/project/libs/
```

总结

1. 环境准备: 确保 JDK、CMake 和必要的构建工具已正确安装

2. 构建选项: 根据需求选择合适的功能开关和优化级别
3. 多平台支持: 使用统一的 CMake 配置支持多种目标平台
4. 问题排查: 遇到问题时, 检查环境变量、依赖库和权限设置

通过遵循本指南, 你应该能够成功编译和集成 Risk Control SDK 到你的项目中。

数据分析

[E05] 数据仓库与处理

E05: 数据仓库与计算引擎

当通过逆向和爬虫采集到海量数据后（例如，数亿条用户行为日志、商品信息），如何存储、管理和分析这些数据，就成了大数据领域的核心问题。本节将介绍主流的数据仓库和分布式计算引擎技术。

1. 数据仓库 (Data Warehouse)

数据仓库是一个用于存储和分析海量结构化、半结构化数据的系统。它与业务数据库（OLTP）不同，其核心目标是支持复杂的分析查询（OLAP）。

a) Hive

- 定位: 基于 Hadoop 的一个数据仓库基础架构。
- 核心思想: Hive 允许你使用标准的 SQL 语言 来查询存储在 Hadoop 分布式文件系统 (HDFS) 上的大规模数据集。它将 SQL 查询翻译成 MapReduce、Tez 或 Spark 任务来执行。
- 元数据 (Metastore): Hive 的核心是其元数据存储。它记录了"表"的结构 (列名、数据类型) 与 HDFS 上的文件 (如 CSV, Parquet, ORC 文件) 之间的映射关系。本质上，Hive 提供了一种"给文件系统套上一个结构化外壳"的能力。
- 适用场景:
 - 对海量 (TB/PB 级) 的、非实时的数据进行离线分析和 ETL (提取、转换、加载)。
 - 构建企业级的数据仓库，为数据分析师和 BI 报表提供统一的 SQL 查询入口。

b) HBase

- 定位: 一个分布式的、可伸缩的、面向列的 NoSQL 数据库。它构建在 HDFS 之上，并模仿 Google Bigtable 的设计。
- 核心特点:
 - 海量存储: 专为存储数十亿行、数百万列的超大规模稀疏数据集而设计。
 - 实时读写: 与 Hive 主要用于离线批量分析不同，HBase 的核心优势在于支持对海量数据的低延迟随机读写。
 - 面向列: 数据按列族 (Column Family) 组织。一个列族中的所有列在物理上存储在一起，这使得对特定列的读取非常高效。
 - 无模式 (Schemaless): 你可以随时向一个列族中添加新的列，而无需预先定义表结构。
- 适用场景:
 - 需要对海量数据进行实时、随机访问的场景，例如用户画像系统、实时推荐引擎的特征库、监控数据的存储。
 - 作为数据采集系统的"落地层"，接收实时写入的数据流，然后由 Hive 或 Spark 进行后续的批量分析。

Hive vs. HBase

特性	Hive	HBase
数据库类型	数据仓库 (SQL on Hadoop)	NoSQL 数据库 (面向列)
核心用途	批量分析 (OLAP)	实时随机读写 (OLTP)
延迟	高 (分钟级)	低 (毫秒级)
数据模型	结构化	半结构化/无模式 (稀疏表)
语言	SQL (HiveQL)	Java API, Shell, Thrift/REST

2. 分布式计算引擎

计算引擎负责实际执行数据处理任务。现代计算引擎通过在内存中进行计算，极大地提升了处理速度。

a) Spark

- 定位: 一个快速、通用、可扩展的分布式计算引擎。
- 核心概念: RDD (弹性分布式数据集): Spark 的基础数据结构。它是一个不可变的、被分区到集群中多个节点上的元素集合，支持丰富的转换 (`map`, `filter`, `join`) 和行动 (`count`, `collect`, `save`) 操作。RDD 的"弹性"体现在其血缘关系 (Lineage)，任何分区的丢失都可以根据其转换历史被重新计算出来。
- DataFrame & Spark SQL: 在 RDD 之上，Spark 提供了更高级的 DataFrame API，它将数据组织成带有命名列的二维表，类似于关系型数据库的表。这使得你可以使用 Spark SQL 来进行结构化数据处理，并且 Spark 的 Catalyst 优化器会自动对你的查询进行优化。
- 生态系统:
 - Spark Streaming: 用于处理实时数据流 (Micro-batching)。
 - MLlib: 提供了一套丰富的机器学习算法库。
 - GraphX: 用于图计算。
- 适用场景:
 - 需要高性能的、迭代式的批量数据处理和机器学习任务。
 - 统一批处理和流处理。

b) Flink

- 定位: 一个以真正的流处理 (True Streaming) 为核心的分布式计算引擎。

- 核心特点:

- 流为核心: Flink 的设计哲学是"一切皆是流", 批量计算被看作是流计算的一个特例。它能够以事件驱动的方式, 逐条处理数据, 实现极低的延迟。
- 状态管理与窗口: Flink 提供了强大的状态管理能力, 允许你在流处理中维护和更新状态(例如, 一个用户的累计消费金额)。它还支持灵活的窗口操作(如滚动窗口、滑动窗口、会话窗口), 用于对无界数据流进行聚合分析。
- 高吞吐与低延迟: 专为低延迟、高吞吐的实时计算场景设计。

- 适用场景:

- 对实时性要求极高的场景, 如实时风控、实时推荐、实时监控大盘。
- 需要进行复杂事件处理(CEP)的场景。

Spark vs. Flink

特性	Spark	Flink
核心模型	批处理(Batch)	流处理(Streaming)
流处理方式	微批次(Micro-batch)	逐条处理(Per-event)
延迟	秒级	毫秒级
窗口	基于时间的窗口	灵活的窗口(时间、计数、会话)
生态	更成熟, 社区更庞大	快速发展, 在实时计算领域是事实标准

总结: 如果你的主要任务是离线分析和机器学习, **Spark** 是一个更通用、更成熟的选择。如果你的核心是需要亚秒级响应的实时计算, **Flink** 则是更专业的工具。在许多现代数据平台中, 二者往往会共存, 分别处理不同的任务。

[E06] Apache Flink

E06: Apache Flink 实时流处理

Apache Flink 是一个分布式流处理框架，专为低延迟、高吞吐量的实时数据处理而设计。

目录

1. Flink 架构
2. 核心概念
3. DataStream API
4. 状态管理
5. 时间与窗口
6. 容错机制
7. 性能调优
8. 面试要点

Flink 架构

集群架构

```
└── ResourceManager
    └── JobMaster
        ↓
        TaskManager1 TaskManager2 TaskManager3
        ├── Task Slot1  └── Task Slot1  └── Task Slot1
        ├── Task Slot2  └── Task Slot2  └── Task Slot2
        └── Task Slot3  └── Task Slot3  └── Task Slot3
```

| TaskManager | 执行具体任务, 管理内存和网络 | | Dispatcher | 接收作业提交, 启动 JobMaster | | ResourceManager | 管理 TaskManager 资源 | | JobMaster | 管理单个作业的执行 |

运行时架构

```
Operator1 → Operator2 → Operator3
```

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> stream = env
    .socketTextStream("localhost", 9999)
    .flatMap(new Tokenizer())
    .keyBy(value -> value.f0)
    .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
    .sum(1);

env.execute("Word Count");
```

```
DataStream<Integer> intStream;

// 2. 元组类型
DataStream<Tuple2<String, Integer>> tupleStream;

// 3. POJO 类型
public class WordCount {
    public String word;
    public int count;
    // constructors, getters, setters
}
DataStream<WordCount> pojoStream;

// 4. Row 类型 (动态)
DataStream<Row> rowStream;
```

StreamExecutionEnvironment env
StreamExecutionEnvironment.getExecutionEnvironment();

```
// 1. Create from collection DataStream fromCollection = env.fromCollection(Arrays.asList("a", "b", "c"));

// 2. 从文件系统 DataStream fromFile = env.readTextFile("path/to/file");

// 3. 从 Kafka Properties props = new Properties(); props.setProperty("bootstrap.servers", "localhost:9092"); props.setProperty("group.id", "test");

DataStream fromKafka = env.addSource( new FlinkKafkaConsumer<>("topic", new SimpleStringSchema(), props));

// 4. 自定义数据源 DataStream customSource = env.addSource(new CustomSourceFunction());

public class CustomSourceFunction implements SourceFunction { private volatile boolean running = true;

@Override public void run(SourceContext ctx) throws Exception { while (running) { ctx.collect("data-" + System.currentTimeMillis()); Thread.sleep(1000); } }

@Override public void cancel() { running = false; } }
```

```
DataStream<String> mapped = input.map(String::toUpperCase);

// 2. FlatMap - 一对多转换
DataStream<String> flatMapped = input.flatMap(
    (String line, Collector<String> out) -> {
        for (String word : line.split(" ")) {
            out.collect(word);
        }
    });

// 3. Filter - 过滤
DataStream<String> filtered = input.filter(s -> s.startsWith("error"));

// 4. KeyBy - 分组
KeyedStream<Tuple2<String, Integer>, String> keyed =
    tupleStream.keyBy(value -> value.f0);

// 5. Reduce - 聚合
DataStream<Tuple2<String, Integer>> reduced =
    keyed.reduce((a, b) -> new Tuple2<>(a.f0, a.f1 + b.f1));

// 6. Aggregate - 自定义聚合
DataStream<Double> aggregated = keyed
    .aggregate(new AverageAggregate());

public class AverageAggregate implements AggregateFunction<Tuple2<String, Integer>,
    Tuple2<Long, Long>, Double> {
    @Override
    public Tuple2<Long, Long> createAccumulator() {
        return new Tuple2<>(0L, 0L);
    }

    @Override
    public Tuple2<Long, Long> add(Tuple2<String, Integer> value, Tuple2<Long, Long>
        accumulator) {
        return new Tuple2<>(accumulator.f0 + value.f1, accumulator.f1 + 1L);
    }

    @Override
    public Double getResult(Tuple2<Long, Long> accumulator) {
        return ((double) accumulator.f0) / accumulator.f1;
    }

    @Override
    public Tuple2<Long, Long> merge(Tuple2<Long, Long> a, Tuple2<Long, Long> b) {
        return new Tuple2<>(a.f0 + b.f0, a.f1 + b.f1);
    }
}

// 2. 写入文件 stream.writeAsText("path/to/output");
```

```
// 3. 写入 Kafka Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost:9092");

stream.addSink(new FlinkKafkaProducer<>("output-topic", new SimpleStringSchema(),
props));

// 4. 写入数据库 stream.addSink(new CustomSinkFunction());

public class CustomSinkFunction extends RichSinkFunction { private Connection
connection;

@Override public void open(Configuration parameters) throws Exception { // 初始化数
据库连接 connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/
test", "user", "password"); }

@Override public void invoke(String value, Context context) throws Exception { // 执行
插入操作 PreparedStatement stmt = connection.prepareStatement("INSERT INTO table
VALUES (?)"); stmt.setString(1, value); stmt.executeUpdate(); }

@Override public void close() throws Exception { if (connection != null)
{ connection.close(); } } }
```

```
```java
public class StatefulMap extends RichMapFunction<Tuple2<String, Integer>,
Tuple2<String, Integer>> {
private ValueState<Integer> sumState;

@Override
public void open(Configuration config) {
ValueStateDescriptor<Integer> descriptor =
new ValueStateDescriptor<>("sum", Integer.class);
sumState = getRuntimeContext().getState(descriptor);
}

@Override
public Tuple2<String, Integer> map(Tuple2<String, Integer> input) throws Exception {
Integer currentSum = sumState.value();
if (currentSum == null) {
currentSum = 0;
}
currentSum += input.f1;
sumState.update(currentSum);

return new Tuple2<>(input.f0, currentSum);
}
}
```

```
private ListState checkpointerState;
```

```
@Override public void snapshotState(FunctionSnapshotContext context) throws
Exception { checkpointerState.clear(); for (String element : bufferedElements)
{ checkpointerState.add(element); } }
```

```
@Override public void initializeState(FunctionInitializationContext context) throws
Exception { ListStateDescriptor descriptor = new ListStateDescriptor<>("buffered-
elements", String.class);
```

```
checkpointerState = context.getOperatorStateStore().getListState(descriptor);
```

```
if (context.isRestored()) { for (String element : checkpointerState.get())
{ bufferedElements.add(element); } } }
```

```
// 2. FsStateBackend (生产推荐)
env.setStateBackend(new FsStateBackend("hdfs://namenode:port/flink-checkpoints"));

// 3. RocksDBStateBackend (大状态)
env.setStateBackend(new RocksDBStateBackend("hdfs://namenode:port/flink-
checkpoints"));

// 配置
env.enableCheckpointing(60000); // 60 秒 checkpoint 一次
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(30000);
env.getCheckpointConfig().setCheckpointTimeout(600000);
```

// 1. Processing Time (处理时间)

```
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
```

// 2. Event Time (事件时间)

```
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

// 3. Watermark (水位线)

```
stream.assignTimestampsAndWatermarks(WatermarkStrategy.forBoundedOutOfOrderness(Duration.ofSeconds(5))
 .withTimestampExtractor((timestamp) -> event.getTimestamp()));
```

```
stream.keyBy(...)
 .window(TumblingEventTimeWindows.of(Time.minutes(5)))
 .sum(1);
```

```
// 滑动窗口 (Sliding Window)
stream.keyBy(...)
 .window(SlidingEventTimeWindows.of(Time.minutes(10), Time.minutes(2)))
 .sum(1);
```

```
// 会话窗口 (Session Window)
stream.keyBy(...)
 .window(EventTimeSessionWindows.withGap(Time.minutes(30)))
 .sum(1);
```

```
.countWindow(100).sum(1);
```

```
// 滑动计数窗口 stream.keyBy(...).countWindow(100, 10).sum(1);
```

```
.window(TumblingEventTimeWindows.of(Time.minutes(5)))
.reduce(new SumReduceFunction());

// 2. AggregateFunction
stream.keyBy(...)
.window(TumblingEventTimeWindows.of(Time.minutes(5)))
.aggregate(new AverageAggregateFunction());

// 3. ProcessWindowFunction
stream.keyBy(...)
.window(TumblingEventTimeWindows.of(Time.minutes(5)))
.process(new MyProcessWindowFunction());

public class MyProcessWindowFunction extends ProcessWindowFunction<Tuple2<String,
Integer>, String, String, TimeWindow> {
@Override
public void process(String key, Context context, Iterable<Tuple2<String, Integer>>
elements, Collector<String> out) {
int count = 0;
for (Tuple2<String, Integer> element : elements) {
count++;
}
out.collect("Window: " + context.window() + " count: " + count);
}
}
```

```
// EnabledCheckpoint env.enableCheckpointing(60000,
CheckpointingMode.EXACTLY_ONCE);

// ConfigCheckpoint CheckpointConfig config = env.getCheckpointConfig();
config.setMinPauseBetweenCheckpoints(30000);
config.setCheckpointTimeout(600000); config.setMaxConcurrentCheckpoints(1);
config.enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

// 重启 Strategy env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3, // 重
启次数 Time.of(10, TimeUnit.SECONDS) // 重启间隔));

env.setRestartStrategy(RestartStrategies.exponentialDelayRestart(Time.milliseconds(1),
Time.milliseconds(1000),

1. 2, // 指数基数 Time.milliseconds(5000), // 最大 Latency
2. 1 // 抖动因子));
```

```
从 Savepoint 恢复
bin/flink run -s <savepointPath> <jarFile>

DeleteSavepoint
bin/flink savepoint -d <savepointPath>
```

```
// 1. 全局并行度 env.setParallelism(4);

// 2. 算子并行度 stream.map(...).setParallelism(2);

// 3. Slot 共享组 stream.map(...).slotSharingGroup("group1");
```

```
taskmanager.memory.flink.size: 3g
taskmanager.memory.network.fraction: 0.1
taskmanager.memory.managed.fraction: 0.4
```

```
// 批量传输 env.setBufferTimeout(100);
```

```
backend.setPredefinedOptions(PredefinedOptions.SPINNING_DISK_OPTIMIZED);
backend.setDbStoragePath("/tmp/rocksdb");
env.setStateBackend(backend);
```

|:---|:---|:---| | 处理模型 | 真正的流处理 | 微批处理 | | 延迟 | 毫秒级 | 秒级 | | 吞吐量 | 高 | 很高 | | 状态管理 | 原生支持 | 有限支持 | | 容错 | Checkpoint | RDD lineage | | 反压 | 原生支持 | 有限支持 |

## # 2. 反压 (Backpressure) 机制

- 问题: 下游处理速度跟不上上游产生速度

- Flink 解决方案:

1. 信用机制: 基于信用的流量控制
2. 缓冲池: 动态调整缓冲池大小

3. 网络栈: TCP 流量控制
4. 监控: Web UI 显示反压情况

## # 3. Exactly-Once 语义保证

- 组件:

1. Source: 可重放 (如 Kafka offset)
2. 内部处理: Checkpoint 机制
3. Sink: 两阶段提交或幂等写入

```
// 两阶段提交 Sink 示例
public class TwoPhaseCommitSink extends TwoPhaseCommitSinkFunction<String,
Transaction, Void> {
@Override
protected Transaction beginTransaction() throws Exception {
return new Transaction();
}

@Override
protected void invoke(Transaction transaction, String value, Context context) throws
Exception {
transaction.add(value);
}

@Override
protected void preCommit(Transaction transaction) throws Exception {
transaction.flush();
}

@Override
protected void commit(Transaction transaction) {
transaction.commit();
}

@Override
protected void abort(Transaction transaction) {
transaction.rollback();
}
}
```

1. Watermark: 事件时间窗口

2. Processing Time: 处理时间窗口
3. 元素计数: 计数窗口
4. 自定义: 用户定义的触发器

## # 5. 状态管理最佳实践

1. 选择合适的状态类型: ValueState vs ListState vs MapState
2. 设置状态 TTL: 避免状态无限增长
3. 选择合适的状态后端: 内存 vs 文件系统 vs RocksDB
4. 状态大小监控: 及时发现状态膨胀

```
// SetStatusTTL
StateTtlConfig ttlConfig = StateTtlConfig
 .newBuilder(Time.days(7))
 .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)
 .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
 .build();

ValueStateDescriptor<String> descriptor = new ValueStateDescriptor<>("my-state",
String.class);
descriptor.enableTimeToLive(ttlConfig);
```

1. \*\*检测故障\*\*: JobManager 监控 TaskManager 心跳
2. \*\*重启任务\*\*: 根据重启策略重启失败任务
3. \*\*恢复状态\*\*: 从最近的 Checkpoint 恢复状态
4. \*\*重放数据\*\*: Source 重放 Checkpoint 之后的数据
5. \*\*继续处理\*\*: 恢复正常处理流程

---

## [E07] HBase 数据库

# E07: HBase 分布式 NoSQL 数据库

---

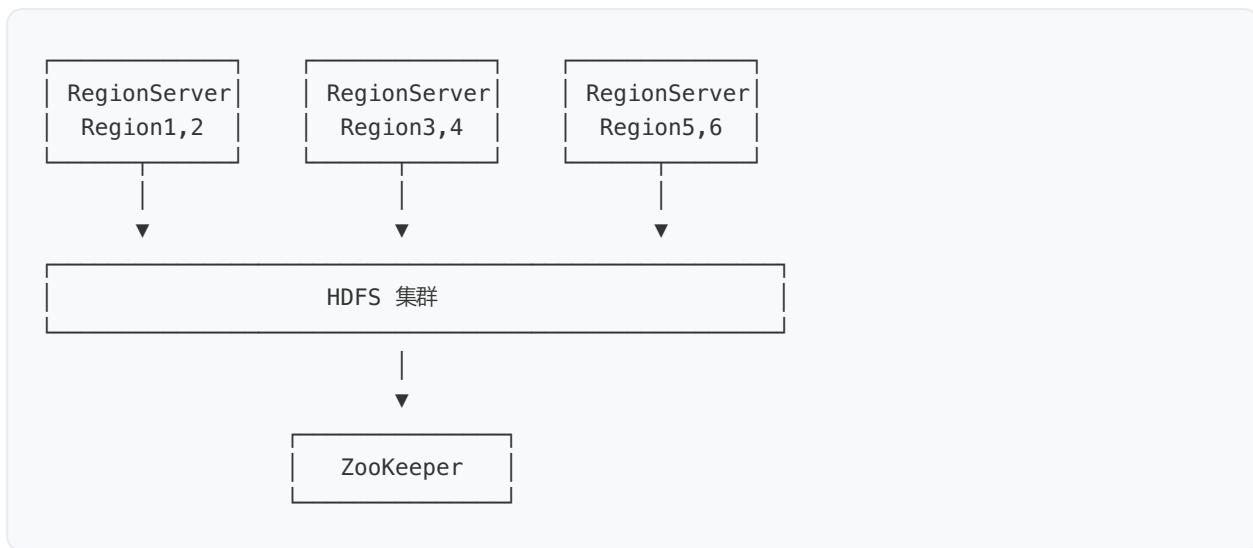
Apache HBase 是一个分布式、面向列的开源数据库，基于 Google Bigtable 论文实现，运行在 HDFS 之上。

## 目录

- [1. HBase 架构](#)
  - [2. 数据模型](#)
  - [3. Shell 操作](#)
  - [4. Java API](#)
  - [5. 性能优化](#)
  - [6. 集群管理](#)
  - [7. 面试要点](#)
-

## HBase 架构

### 核心组件



组件	说明
HMaster	管理 RegionServer, 处理表的创建/删除, Region 分配
RegionServer	处理数据读写请求, 管理多个 Region
Region	表的水平分片, 包含一定范围的行
ZooKeeper	协调服务, 存储元数据, 故障检测
HDFS	底层存储系统

## Region 详细结构

```
Region
└── Store (列族)
 ├── MemStore (内存缓冲)
 └── HFile (磁盘文件)
 ├── Data Block
 ├── Index Block
 └── Bloom Filter
└── WAL (Write-Ahead Log)
```

## 写入流程

1. Client → RegionServer: 写请求
2. WAL: 先写预写日志 (保证数据持久性)
3. MemStore: 数据写入内存
4. Flush: MemStore 达到阈值时刷写到 HFile
5. Compaction: 定期合并 HFile

## 读取流程

1. Client → RegionServer: 读请求
2. MemStore: 先查内存中的数据
3. BlockCache: 查询缓存的 HFile 块
4. HFile: 从磁盘读取数据
5. Merge: 合并多个来源的数据返回

## 数据模型

### 逻辑视图

```
Table: user_info
└── Row Key: user001
 ├── Column Family: basic_info
 │ ├── name:zhang_san (timestamp:1234567890)
 │ └── age:25 (timestamp:1234567891)
 └── Column Family: contact_info
 ├── email:zhang@example.com (timestamp:1234567892)
 └── phone:13800138000 (timestamp:1234567893)
└── Row Key: user002
 └── ...
```

### 物理存储

```
basic_info Store:
user001:name:1234567890 → zhang_san
user001:age:1234567891 → 25
user002:name:1234567900 → li_si

contact_info Store:
user001:email:1234567892 → zhang@example.com
user001:phone:1234567893 → 13800138000
```

## 核心概念

概念	说明
Row Key	行键, 表的主键, 按字典序排序
Column Family	列族, 列的逻辑分组
Column Qualifier	列限定符, 列族下的具体列
Cell	单元格, 由(row, column family, column qualifier, timestamp)确定
Timestamp	时间戳, 同一 Cell 的多个版本

## Shell 操作

### 连接与基本操作

```
启动 HBase Shell
hbase shell

查看状态
status
version

查看集群信息
whoami
```

## 命名空间管理

```
创建命名空间
create_namespace 'my_namespace'

列出命名空间
list_namespace

删除命名空间
drop_namespace 'my_namespace'
```

## 表操作

```
创建表 (带配置)
create 'user_info',
 {NAME => 'basic_info', VERSIONS => 3, TTL => 2592000},
 {NAME => 'contact_info', COMPRESSION => 'SNAPPY'}

列出表
list

查看表结构
describe 'user_info'

禁用表
disable 'user_info'

启用表
enable 'user_info'

删除表
drop 'user_info'

修改表结构
alter 'user_info', {NAME => 'basic_info', VERSIONS => 5}
```

## 数据操作

```
插入数据
put 'user_info', 'user001', 'basic_info:name', 'zhang_san'
put 'user_info', 'user001', 'basic_info:age', '25'
put 'user_info', 'user001', 'contact_info:email', 'zhang@example.com'

查询单行
get 'user_info', 'user001'

查询指定列族
get 'user_info', 'user001', 'basic_info'

查询指定列
get 'user_info', 'user001', 'basic_info:name'

扫描表
scan 'user_info'

条件扫描
scan 'user_info', {STARTROW => 'user001', ENDROW => 'user999'}
scan 'user_info', {FILTER => "SingleColumnValueFilter('basic_info', 'age', >=, 'binary:18')"}

删除数据
delete 'user_info', 'user001', 'basic_info:age'

删除行
deleteall 'user_info', 'user001'

计数
count 'user_info'
```

## Java API

### 连接配置

```
// 配置连接
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "node1,node2,node3");
conf.set("hbase.zookeeper.property.clientPort", "2181");

// 创建连接
Connection connection = ConnectionFactory.createConnection(conf);
Admin admin = connection.getAdmin();
```

### 表管理

```
// 创建表
TableName tableName = TableName.valueOf("user_info");
HTableDescriptor tableDesc = new HTableDescriptor(tableName);

// 添加列族
HColumnDescriptor basicInfo = new HColumnDescriptor("basic_info");
basicInfo.setMaxVersions(3);
basicInfo.setTimeToLive(30 * 24 * 60 * 60); // 30 天 TTL

HColumnDescriptor contactInfo = new HColumnDescriptor("contact_info");
contactInfo.setCompressionType(Compression.Algorithm.SNAPPY);

tableDesc.addFamily(basicInfo);
tableDesc.addFamily(contactInfo);

// 创建表
admin.createTable(tableDesc);
```

## 数据操作

```
Table table = connection.getTable(TableName.valueOf("user_info"));

// 插入数据
Put put = new Put(Bytes.toBytes("user001"));
put.addColumn(Bytes.toBytes("basic_info"), Bytes.toBytes("name"),
Bytes.toBytes("zhang_san"));
put.addColumn(Bytes.toBytes("basic_info"), Bytes.toBytes("age"), Bytes.toBytes("25"));
table.put(put);

// 批量插入
List<Put> puts = new ArrayList<>();
for (int i = 0; i < 1000; i++) {
 Put batchPut = new Put(Bytes.toBytes("user" + String.format("%03d", i)));
 batchPut.addColumn(Bytes.toBytes("basic_info"), Bytes.toBytes("name"),
Bytes.toBytes("user" + i));
 puts.add(batchPut);
}
table.put(puts);

// 查询数据
Get get = new Get(Bytes.toBytes("user001"));
get.addFamily(Bytes.toBytes("basic_info"));
Result result = table.get(get);

// 解析结果
for (Cell cell : result.listCells()) {
 String family = Bytes.toString(CellUtil.cloneFamily(cell));
 String qualifier = Bytes.toString(CellUtil.cloneQualifier(cell));
 String value = Bytes.toString(CellUtil.cloneValue(cell));
 System.out.println(family + ":" + qualifier + " = " + value);
}

// 扫描数据
Scan scan = new Scan();
scan.setStartRow(Bytes.toBytes("user001"));
scan.setStopRow(Bytes.toBytes("user999"));
scan.addFamily(Bytes.toBytes("basic_info"));

ResultScanner scanner = table.getScanner(scan);
for (Result res : scanner) {
 // 处理结果
}
scanner.close();
```

## 过滤器

```
// 单列值过滤器
SingleColumnValueFilter filter = new SingleColumnValueFilter(
 Bytes.toBytes("basic_info"),
 Bytes.toBytes("age"),
 CompareFilter.CompareOp.GREATER_OR_EQUAL,
 Bytes.toBytes("18")
);

// 前缀过滤器
PrefixFilter prefixFilter = new PrefixFilter(Bytes.toBytes("user00"));

// 组合过滤器
FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL);
filterList.addFilter(filter);
filterList.addFilter(prefixFilter);

scan.setFilter(filterList);
```

## 性能优化

### Row Key 设计

```
// 1. 避免热点: 使用散列前缀
String rowKey = MD5Hash.digest(userId).toString().substring(0, 2) + "_" + userId;

// 2. 时间倒序: 便于查询最新数据
String rowKey = userId + "_" + (Long.MAX_VALUE - timestamp);

// 3. 组合 Key: 支持多维查询
String rowKey = region + "_" + userId + "_" + timestamp;
```

## 列族设计

```
// 合理设计: 按访问模式分组
create 'user_profile',
{NAME => 'profile', COMPRESSION => 'SNAPPY'}, // 用户基本信息
{NAME => 'behavior', TTL => 604800} // 用户行为数据(7天 TTL)

// 避免: 列族过多
// 避免: 不同访问模式的列放在同一列族
```

## 批量操作

```
List<Put> puts = new ArrayList<>();
for (UserData user : userData) {
 Put put = createPut(user);
 puts.add(put);

 // 批量提交
 if (puts.size() >= 1000) {
 table.put(puts);
 puts.clear();
 }
}
// 提交剩余数据
if (!puts.isEmpty()) {
 table.put(puts);
}
```

## 缓存配置

```
// 启用 Block Cache
HColumnDescriptor family = new HColumnDescriptor("data");
family.setBlockCacheEnabled(true);
family.setCacheBloomsOnWrite(true);
family.setCacheDataOnWrite(true);
family.setCacheIndexesOnWrite(true);
```

## 集群管理

### Region 管理

```
手动分裂 Region
split 'user_info', 'user500'

合并 Region
merge_region 'region1_encoded_name', 'region2_encoded_name'

查看 Region 信息
list_regions 'user_info'
```

### 负载均衡

```
手动触发负载均衡
balancer

查看负载均衡状态
balancer_enabled
```

### Compaction

```
手动触发 Major Compaction
major_compact 'user_info'

手动触发 Minor Compaction
compact 'user_info'

查看压缩状态
compaction_state 'user_info'
```

## 监控

```
查看 RegionServer 信息
list_regionservers

查看表统计信息
list_table_stats 'user_info'
```

## 面试要点

### 1. HBase vs 关系型数据库

特性	HBase	关系型数据库
数据模型	列族模型	关系模型
ACID	行级原子性	完整ACID
扩展性	水平扩展	垂直扩展
查询语言	NoSQL API	SQL
适用场景	大数据读写	复杂事务

### 2. 数据倾斜问题

问题: Region 热点, 某些 Region 访问量过大

解决方案:

1. Row Key 设计: 避免单调递增, 使用散列前缀
2. 预分区: 创建表时预先分区
3. 负载均衡: 定期执行 balance 操作

```
// 预分区示例
byte[][] splits = new byte[10][];
for (int i = 0; i < 10; i++) {
 splits[i] = Bytes.toBytes(String.format("%02d", i));
}
admin.createTable(tableDesc, splits);
```

### 3. 读取性能优化

1. Bloom Filter: 快速判断数据是否存在
2. Block Cache: 缓存热点数据
3. 压缩: 减少存储空间和 IO
4. 预读: 设置合理的扫描缓存

```
// 配置 Bloom Filter
HColumnDescriptor family = new HColumnDescriptor("data");
family.setBloomFilterType(BloomType.ROW);
```

### 4. 写入性能优化

1. 批量写入: 减少 RPC 调用
2. WAL: 根据需要关闭 WAL
3. MemStore: 调整内存大小
4. 压缩: 异步压缩

```
// 关闭 WAL (数据安全性降低)
put.setDurability(Durability.SKIP_WAL);
```

### 5. 故障恢复

1. WAL: 通过预写日志恢复数据
2. Region 迁移: 自动迁移故障节点的 Region
3. ZooKeeper: 监控集群状态, 协调故障恢复

---

#### 4. 数据副本: 依赖 HDFS 的数据副本机制

## 6. 热点问题诊断

```
1. 查看 Region 分布
list_regions 'table_name'

2. 查看 RegionServer 负载
status 'detailed'

3. 分析访问模式
通过日志分析热点 Row Key

4. 重新设计 Row Key
添加散列前缀或使用反向时间戳
```

## [E08] Apache Hive

# E08: Hive 数据仓库

Apache Hive 是基于 Hadoop 的数据仓库工具，可以将结构化的数据文件映射为数据库表，并提供类 SQL 查询功能。

## 目录

1. Hive 架构
2. 数据类型与存储格式
3. DDL 操作
4. DML 操作
5. 分区与分桶
6. 函数与 UDF
7. 性能优化
8. 面试要点

## Hive 架构

### # 核心组件

```
Metastore (Metadata storage)
↓
HDFS (Data storage) + MapReduce/Tez/Spark (Compute engine)
```

---

| Driver | 解析 SQL、生成执行计划、协调执行 | | Execution Engine | 执行引擎  
(MapReduce/Tez/Spark) |

## # 工作流程

1. SQL 解析: 词法分析 → 语法分析 → 语义分析
  2. 逻辑计划: 生成逻辑执行计划
  3. 物理计划: 转换为 MapReduce/Tez/Spark 任务
  4. 执行: 提交到 Hadoop 集群执行
-

## 数据类型与存储格式

### # 基本数据类型

类型	描述	示例
TINYINT	1 字节整数	-128 到 127
SMALLINT	2 字节整数	-32,768 到 32,767
INT	4 字节整数	-2,147,483,648 到 2,147,483,647
BIGINT	8 字节整数	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
FLOAT	4 字节浮点数	3.14159
DOUBLE	8 字节浮点数	3.141592653589793
STRING	字符串	'Hello World'
BOOLEAN	布尔值	TRUE/FALSE
TIMESTAMP	时间戳	'2023-01-01 12:00:00'
DATE	日期	'2023-01-01'

## # 复杂数据类型

```
-- Array
ARRAY<data_type>
-- 示例: ARRAY<STRING>

-- Map
MAP<primitive_type, data_type>
-- 示例: MAP<STRING, INT>

-- 结构体
STRUCT<col_name:data_type [COMMENT col_comment], ...>
-- 示例: STRUCT<name:STRING, age:INT>

-- 联合体
UNIONTYPE<data_type, data_type, ...>
```

| SequenceFile | 中 | 中 | 中 | 中间数据 || RCFFile | 高 | 中 | 低 | 列式分析 || ORC | 很高 |  
很高 | 低 | OLAP 分析 || Parquet | 很高 | 很高 | 低 | 跨平台分析 |

## DDL 操作

### # 数据库操作

```
-- 创建数据库
CREATE DATABASE IF NOT EXISTS mydb
COMMENT 'My database'
LOCATION '/user/hive/warehouse/mydb.db';

-- 使用数据库
USE mydb;

-- 显示数据库
SHOW DATABASES;

-- 删除数据库
DROP DATABASE IF EXISTS mydb CASCADE;
```

```
id INT, name STRING, salary DOUBLE, department STRING) STORED AS ORC
TBLPROPERTIES ('orc.compress'='SNAPPY');
```

```
name STRING,
salary DOUBLE,
department STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/data/employee/';
```

```
name STRING, salary DOUBLE) PARTITIONED BY (department STRING, year INT)
STORED AS ORC;
```

```
-- 修改列
ALTER TABLE employee CHANGE salary salary DECIMAL(10,2);

-- 添加分区
ALTER TABLE partitioned_employee ADD PARTITION (department='IT', year=2023);

-- 删除分区
ALTER TABLE partitioned_employee DROP PARTITION (department='IT', year=2023);

-- 重命名表
ALTER TABLE employee RENAME TO emp;
```

- 插入数据 INSERT INTO employee VALUES (1, 'John', 5000.0, 'IT');
- 从查询插入 INSERT INTO employee SELECT id, name, salary, department  
FROM temp\_employee;
- 覆盖插入 INSERT OVERWRITE TABLE employee SELECT \* FROM  
temp\_employee;
- 分区插入 INSERT INTO partitioned\_employee PARTITION(department='IT',  
year=2023) SELECT id, name, salary FROM temp\_employee;
- 动态分区插入 SET hive.exec.dynamic.partition=true; SET  
hive.exec.dynamic.partition.mode=nonstrict;

```
INSERT INTO partitioned_employee PARTITION(department, year) SELECT id, name, salary, department, year FROM temp_employee;
```

-- 聚合查询

```
SELECT department, AVG(salary) as avg_salary
FROM employee
GROUP BY department
HAVING AVG(salary) > 6000;
```

-- 连接查询

```
SELECT e.name, d.dept_name
FROM employee e
JOIN department d ON e.department = d.dept_id;
```

-- 窗口函数

```
SELECT name, salary,
ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as rank
FROM employee;
```

-- 静态分区

```
INSERT INTO partitioned_employee PARTITION(department='IT', year=2023)
SELECT id, name, salary FROM source_table;
```

-- 动态分区

```
INSERT INTO partitioned_employee PARTITION(department, year)
SELECT id, name, salary, department, year FROM source_table;
```

-- 查询特定分区

```
SELECT * FROM partitioned_employee
WHERE department='IT' AND year=2023;
```

```
CREATE TABLE bucketed_employee (id INT, name STRING, salary DOUBLE, department STRING) CLUSTERED BY (id) INTO 4 BUCKETS STORED AS ORC;
```

- 启用分桶 SET hive.enforce.bucketing=true;
- 插入数据（自动分桶） INSERT INTO bucketed\_employee SELECT \* FROM employee;

```
```sql
-- 字符串操作
SELECT
CONCAT(first_name, ' ', last_name) as full_name,
UPPER(name) as upper_name,
LENGTH(name) as name_length,
SUBSTR(name, 1, 3) as name_prefix
FROM employee;
```

```
FROM_UNIXTIME(UNIX_TIMESTAMP()) as current_time, YEAR(hire_date) as hire_year,
DATEDIFF(CURRENT_DATE, hire_date) as days_since_hire FROM employee;
```

```
COUNT(*) as total_count,
SUM(salary) as total_salary,
AVG(salary) as avg_salary,
MAX(salary) as max_salary,
MIN(salary) as min_salary,
STDDEV(salary) as salary_stddev
FROM employee;
```

```
public String evaluate(String input) { if (input == null) return null; return
input.toUpperCase(); } }
```

```
-- 使用 UDF
SELECT my_upper(name) FROM employee;
```

- 选择合适的存储格式 CREATE TABLE optimized_table (col1 STRING, col2 INT)
STORED AS ORC TBLPROPERTIES ('orc.compress'='SNAPPY',
'orc.create.index'='true');

```
SELECT * FROM partitioned_table  
WHERE partition_col = 'value';  
  
-- 避免: 全表扫描  
SELECT * FROM partitioned_table  
WHERE non_partition_col = 'value';
```

- 避免: SELECT _ SELECT _ FROM employee;

```
FROM employee e  
JOIN department d ON e.dept_id = d.id  
WHERE e.salary > 5000;
```

- 启用代价优化器 SET hive.cbo.enable=true;
- 设置合理的 MapReduce 参数 SET mapreduce.job.reduces=10; SET
hive.exec.reducer.bytes.per.reducer=1000000000;
- 启用并行执行 SET hive.exec.parallel=true; SET
hive.exec.parallel.thread.number=8;

---	---	---
数据量	PB级	GB-TB级
延迟	高(秒-分钟)	低(毫秒)
ACID	有限支持	完全支持
索引	有限	丰富
扩展性	水平扩展	垂直扩展
适用场景	离线分析	在线事务

2. 内部表 vs 外部表

特性	内部表	外部表
---	---	---
数据管理	Hive管理	用户管理
删除表	删除元数据和数据	只删除元数据
数据位置	Hive仓库目录	用户指定位置
使用场景	临时数据、中间结果	共享数据、外部数据源

3. 数据倾斜解决方案

```
```sql
-- 1. 增加 Reduce 任务数
SET mapreduce.job.reduces=100;

-- 2. 启用负载均衡
SET hive.groupby.skewindata=true;

-- 3. 使用随机前缀
SELECT /*+ MAPJOIN(b) */ *
FROM (
 SELECT CONCAT(CAST(RAND() * 100 AS INT), '_', key) as new_key, value
 FROM skewed_table
) a
JOIN small_table b ON a.key = b.key;
```

- ◦ 1. 合并小文件 SET hive.merge.mapfiles=true; SET  
      hive.merge.mapredfiles=true; SET hive.merge.size.per.task=256000000;
- ◦ 1. 使用 Concatenate ALTER TABLE table\_name CONCATENATE;
- ◦ 1. 重新组织数据 INSERT OVERWRITE TABLE new\_table SELECT \* FROM  
      old\_table;

4. \*\*分桶\*\*: 对大表使用分桶
5. \*\*索引\*\*: 创建适当的索引
6. \*\*缓存\*\*: 缓存热点数据
7. \*\*并行度\*\*: 调整Map/Reduce任务数
8. \*\*资源\*\*: 合理分配内存和CPU

## [E09] Apache Spark

# E09: Apache Spark 大数据处理

---

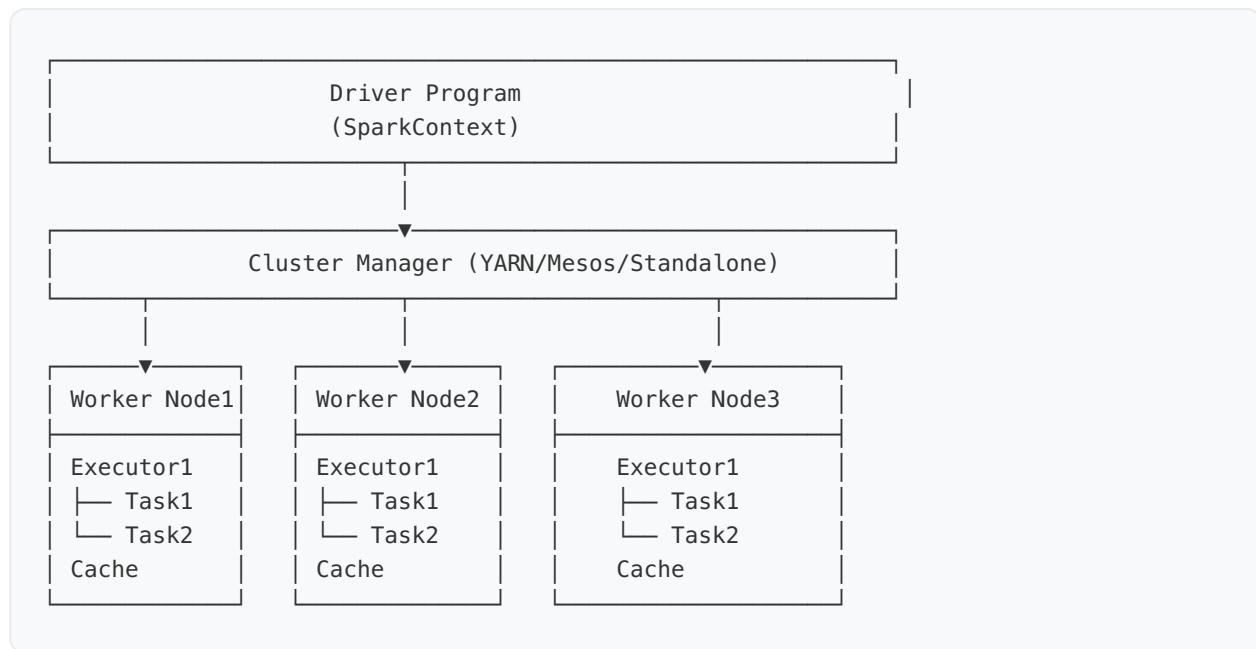
Apache Spark 是一个统一的大数据处理引擎，支持批处理、流处理、机器学习和图计算。

## 目录

- [1. Spark 架构](#)
  - [2. RDD 编程](#)
  - [3. DataFrame & Dataset](#)
  - [4. Spark SQL](#)
  - [5. Spark Streaming](#)
  - [6. 性能优化](#)
  - [7. 面试要点](#)
-

# Spark 架构

## 集群架构



## 核心组件

组件	说明
SparkContext	Spark 程序入口，协调集群资源
Cluster Manager	集群资源管理器
Worker Node	集群中的工作节点
Executor	运行在 Worker 上的进程，执行 Task
Task	最小的工作单元

## 运行流程

1. Driver 创建 SparkContext

- 
2. 资源申请: 向 Cluster Manager 申请资源
  3. 任务调度: DAG Scheduler 将 Job 分解为 Stage 和 Task
  4. 任务分发: Task Scheduler 将 Task 分发到 Executor
  5. 任务执行: Executor 执行 Task 并返回结果
- 

## RDD 编程

### RDD 基本概念

RDD (Resilient Distributed Dataset): 弹性分布式数据集

#### 特性

- 不可变性: RDD 一旦创建不可修改
- 分区性: 数据分布在多个分区
- 容错性: 通过血统(Lineage)恢复丢失数据
- 惰性求值: 只有在 Action 操作时才会执行

### RDD 创建

```
import org.apache.spark.{SparkConf, SparkContext}

val conf = new SparkConf().setAppName("SparkExample").setMaster("local[*]")
val sc = new SparkContext(conf)

// 1. 从集合创建
val rdd1 = sc.parallelize(List(1, 2, 3, 4, 5))
val rdd2 = sc.makeRDD(Array("a", "b", "c"))

// 2. 从外部存储创建
val rdd3 = sc.textFile("hdfs://path/to/file")
val rdd4 = sc.wholeTextFiles("hdfs://path/to/directory")

// 3. 从其他 RDD 创建
val rdd5 = rdd1.map(_ * 2)
```

## Transformation 操作

```
val data = sc.parallelize(List(1, 2, 3, 4, 5))

// 1. map - 一对一转换
val mapped = data.map(_ * 2)

// 2. filter - 过滤
val filtered = data.filter(_ % 2 == 0)

// 3. flatMap - 一对多转换
val words = sc.parallelize(List("hello world", "spark scala"))
val flatMapped = words.flatMap(_.split(" "))

// 4. distinct - 去重
val distincted = data.distinct()

// 5. union - 合并
val rdd1 = sc.parallelize(List(1, 2, 3))
val rdd2 = sc.parallelize(List(4, 5, 6))
val unioned = rdd1.union(rdd2)

// 6. intersection - 交集
val intersected = rdd1.intersection(rdd2)

// 7. groupByKey - 按键分组
val pairs = sc.parallelize(List(("a", 1), ("b", 2), ("a", 3)))
val grouped = pairs.groupByKey()

// 8. reduceByKey - 按键聚合
val reduced = pairs.reduceByKey(_ + _)

// 9. sortByKey - 按键排序
val sorted = pairs.sortByKey()

// 10. join - 连接
val rdd3 = sc.parallelize(List(("a", "x"), ("b", "y")))
val joined = pairs.join(rdd3)
```

## Action 操作

```
val data = sc.parallelize(List(1, 2, 3, 4, 5))

// 1. collect - 收集所有元素到Driver
val collected = data.collect()

// 2. count - 计算元素数
val count = data.count()

// 3. first - 获取第一个元素
val first = data.first()

// 4. take - 获取前 n 个元素
val taken = data.take(3)

// 5. reduce - 聚合所有元素
val sum = data.reduce(_ + _)

// 6. fold - 带初始值的聚合
val folded = data.fold(0)(_ + _)

// 7. aggregate - 复杂聚合
val (totalSum, totalCount) = data.aggregate((0, 0))(
 (acc, value) => (acc._1 + value, acc._2 + 1),
 (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
)

// 8. foreach - 对每个元素执行操作
data.foreach(println)

// 9. saveAsTextFile - 保存为文本文件
data.saveAsTextFile("hdfs://path/to/output")
```

# DataFrame & Dataset

## DataFrame 创建

```
import org.apache.spark.sql.{SparkSession, DataFrame}

val spark = SparkSession.builder()
 .appName("DataFrameExample")
 .master("local[*]")
 .getOrCreate()

import spark.implicits._

// 1. 从 RDD 创建 DataFrame
case class Person(name: String, age: Int, city: String)
val peopleRDD = spark.sparkContext.parallelize(List(
 Person("Alice", 25, "NYC"),
 Person("Bob", 30, "LA"),
 Person("Charlie", 35, "Chicago")
))
val peopleDF = peopleRDD.toDF()

// 2. 从文件创建 DataFrame
val df = spark.read
 .option("header", "true")
 .option("inferSchema", "true")
 .csv("path/to/file.csv")

// 3. 从 JSON 创建
val jsonDF = spark.read.json("path/to/file.json")
```

## DataFrame 操作

```
// 1. 查看数据
df.show()
df.describe().show()

// 2. 选择列
df.select("name", "age").show()
df.select($"name", $"age" + 1).show()

// 3. 过滤
df.filter($"age" > 25).show()
df.where("age > 25").show()

// 4. 分组聚合
df.groupBy("city").count().show()
df.groupBy("city").agg(avg("age"), max("age")).show()

// 5. 排序
df.orderBy($"age".desc).show()
df.sort("name").show()

// 6. 连接
val df2 = spark.createDataFrame(List(
 ("NYC", "NY"),
 ("LA", "CA"),
 ("Chicago", "IL")
)).toDF("city", "state")

df.join(df2, "city").show()

// 7. 窗口函数
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._

val windowSpec = Window.partitionBy("city").orderBy($"age".desc)
df.withColumn("rank", row_number().over(windowSpec)).show()
```

## Dataset 类型安全

```
// 创建 Dataset
val ds: Dataset[Person] = df.as[Person]

// 类型安全操作
val adults = ds.filter(_.age >= 18)
val names = ds.map(_.name)

// 编译时类型检查
// ds.filter(_.salary > 1000) // 编译错误, Person 没有 salary 字段
```

## Spark SQL

### SQL 查询

```
// 注册临时视图
df.createOrReplaceTempView("people")

// SQL 查询
val result = spark.sql("""
 SELECT city, COUNT(*) as count, AVG(age) as avg_age
 FROM people
 WHERE age > 20
 GROUP BY city
 ORDER BY count DESC
""")

result.show()

// 复杂查询
val complexQuery = spark.sql("""
 SELECT name, age, city,
 ROW_NUMBER() OVER (PARTITION BY city ORDER BY age DESC) as rank
 FROM people
""")
```

## 表管理

```
// 列出所有表
spark.catalog.listTables().show()

// 缓存表
spark.catalog.cacheTable("people")
spark.catalog.uncacheTable("people")

// 删除临时视图
spark.catalog.dropTempView("people")
```

# Spark Streaming

## DStream 编程

```
import org.apache.spark.streaming.{StreamingContext, Seconds}

val ssc = new StreamingContext(spark.sparkContext, Seconds(1))

// 1. 从 socket 创建流
val lines = ssc.socketTextStream("localhost", 9999)

// 2. 转换操作
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// 3. 输出操作
wordCounts.print()

// 启动流处理
ssc.start()
ssc.awaitTermination()
```

## 窗口操作

```
// 窗口聚合
val windowedWordCounts = pairs.reduceByKeyAndWindow(
 (a: Int, b: Int) => a + b, // reduce 函数
 (a: Int, b: Int) => a - b, // inverse reduce 函数
 Seconds(30), // 窗口长度
 Seconds(10) // 滑动间隔
)

// 状态更新
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
 val newCount = newValues.sum + runningCount.getOrElse(0)
 Some(newCount)
}

val stateDstream = pairs.updateStateByKey[Int](updateFunction)
```

## Structured Streaming

```
// 创建流式 DataFrame
val df = spark
 .readStream
 .format("socket")
 .option("host", "localhost")
 .option("port", 9999)
 .load()

// 处理流数据
val words = df.as[String].flatMap(_.split(" "))
val wordCounts = words.groupBy("value").count()

// 输出结果
val query = wordCounts.writeStream
 .outputMode("complete")
 .format("console")
 .trigger(Trigger.ProcessingTime("10 seconds"))
 .start()

query.awaitTermination()
```

## 性能优化

### 缓存策略

```
import org.apache.spark.storage.StorageLevel

// 1. 基本缓存
val cachedRDD = rdd.cache() // MEMORY_ONLY
val persistedRDD = rdd.persist(StorageLevel.MEMORY_AND_DISK)

// 2. DataFrame 缓存
df.cache()
df.persist(StorageLevel.MEMORY_AND_DISK_SER)

// 3. 不同存储级别
StorageLevel.MEMORY_ONLY // 仅内存
StorageLevel.MEMORY_AND_DISK // 内存+磁盘
StorageLevel.MEMORY_ONLY_SER // 内存序列化
StorageLevel.DISK_ONLY // 仅磁盘
StorageLevel.MEMORY_AND_DISK_2 // 内存+磁盘, 2副本
```

### 分区优化

```
// 1. 调整分区数
val repartitioned = rdd.repartition(4) // 增加分区
val coalesced = rdd.coalesce(2) // 减少分区

// 2. 自定义分区器
class CustomPartitioner(numPartitions: Int) extends Partitioner {
 override def numPartitions: Int = numPartitions

 override def getPartition(key: Any): Int = {
 key.hashCode() % numPartitions
 }
}

val partitioned = pairs.partitionBy(new CustomPartitioner(4))

// 3. 数据本地性
val localData = sc.textFile("hdfs://path", minPartitions = 4)
```

## 广播变量与累加器

```
// 1. 广播变量
val broadcastVar = sc.broadcast(List(1, 2, 3))
val result = rdd.map(x => x * broadcastVar.value.sum)

// 2. 累加器
val accum = sc.longAccumulator("My Accumulator")
rdd.foreach(x => accum.add(x))
println(s"Accumulator value: ${accum.value}")

// 3. 自定义累加器
class VectorAccumulator extends AccumulatorV2[Vector, Vector] {
 private var _sum = Vector.zeros(3)

 override def isZero: Boolean = _sum == Vector.zeros(3)
 override def copy(): VectorAccumulator = new VectorAccumulator
 override def reset(): Unit = _sum = Vector.zeros(3)
 override def add(v: Vector): Unit = _sum += v
 override def merge(other: AccumulatorV2[Vector, Vector]): Unit = {
 _sum += other.asInstanceOf[VectorAccumulator]._sum
 }
 override def value: Vector = _sum
}
```

## SQL 优化

```
// 1. 启用 CBO (基于成本的优化)
spark.conf.set("spark.sql.cbo.enabled", "true")

// 2. 启用代码生成
spark.conf.set("spark.sqlcodegen.wholeStage", "true")

// 3. 广播 Join 优化
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")

// 4. 谓词下推
val optimizedDF = df.filter($"age" > 18) // Filter 下推到数据源
```

## 面试要点

### 1. RDD vs DataFrame vs Dataset

特性	RDD	DataFrame	Dataset
类型安全	编译时	运行时	编译时
性能优化	无	Catalyst优化器	Catalyst优化器
API风格	函数式	关系型	类型安全的关系型
内存使用	Java对象	二进制格式	二进制格式
序列化	Java序列化	自定义编码器	自定义编码器

### 2. Spark 内存管理

#### 内存分配

```
Spark 内存 = 堆内存 × spark.memory.fraction (默认 0.6)
├── 存储内存 (Storage) × 0.5
│ └── 用于缓存 RDD、广播变量
├── 执行内存 (Execution) × 0.5
│ └── 用于 Shuffle、Join、Sort、Aggregation
└── 用户内存 (剩余部分)
 └── 用于用户自定义数据结构
```

## 配置示例

```
1. Executor 内存
--executor-memory 4g

2. 内存分配比例
--conf spark.memory.fraction=0.6
--conf spark.memory.storageFraction=0.5

3. 序列化
--conf spark.serializer=org.apache.spark.serializer.KryoSerializer
```

## 3. Shuffle 优化

```
// 1. 预分区
val partitioned = rdd.partitionBy(new HashPartitioner(100))

// 2. 调整并行度
spark.conf.set("spark.sql.shuffle.partitions", "400")

// 3. 启用外部排序
spark.conf.set("spark.sql.execution.useObjectHashAggregateExec", "false")
```

## 4. 数据倾斜处理

```
val skewedRDD = sc.parallelize(List(("key1", 1), ("key1", 1), /* 很多 key1 */, ("key2", 1)))

// 加盐处理
val saltedRDD = skewedRDD.map { case (key, value) =>
 val salt = Random.nextInt(10)
 (s"${key}_${salt}", value)
}

val result = saltedRDD.reduceByKey(_ + _)
 .map { case (saltedKey, value) =>
 val originalKey = saltedKey.split("_")(0)
 (originalKey, value)
 }
 .reduceByKey(_ + _)
```

## 5. 两阶段聚合

```
// 第一阶段: 局部聚合
val localAgg = rdd.mapPartitions { iter =>
 iter.toList.groupBy(_._1).map { case (key, values) =>
 (key, values.map(_._2).sum)
 }.toIterator
}

// 第二阶段: 全局聚合
val globalAgg = localAgg.reduceByKey(_ + _)
```

## 6. 资源配置建议

```
CPU 密集型
--num-executors 10 --executor-cores 5 --executor-memory 2g

内存密集型
--num-executors 5 --executor-cores 2 --executor-memory 8g

平衡型
--num-executors 15 --executor-cores 3 --executor-memory 4g
```

## 7. 最佳实践

```
// 推荐
rdd.filter(...).map(...) // 先过滤再转换

// 避免
rdd.map(...).filter(...) // 先转换再过滤

// 减少 Shuffle
val broadcastVar = sc.broadcast(smallData)
largeRDD.map(x => x + broadcastVar.value) // 使用广播变量代替 join
```

## 8. 数据存储格式

```
// 使用列式存储
df.write
 .mode("overwrite")
 .option("compression", "snappy")
 .parquet("path/to/output") // Parquet 格式

// 分区存储
df.write
 .partitionBy("year", "month")
 .parquet("path/to/partitioned/output")
```

# 附录

---

## [X01] GitHub 开源项目

# X01: 逆向工程 GitHub 开源项目

本列表旨在收集和分类逆向工程领域的优秀开源项目，方便查阅和学习。

## 目录

- 逆向工程领域相关的 GitHub 开源项目
  - 本列表旨在收集和分类逆向工程领域的优秀开源项目，方便查阅和学习。
  - 目录
    - 1. 动态分析与插桩工具
    - 2. 反汇编器与反编译器
    - 3. 调试器
    - 4. 静态分析与二进制分析
    - 5. android 平台
    - 6. 多平台与通用工具
    - 7. Hex 编辑器
    - 8. 脱壳与反混淆
    - 9. 固件分析
    - 10. Apple 平台 (iOS/macOS)
    - 11. 其他与资源

---

## 1. 动态分析与插桩工具

项目	Star 数量	描述
<a href="#">frida/frida</a>	14.5k	跨平台动态插桩框架，支持 Windows, macOS, Linux, iOS, Android, 和 QNX。
<a href="#">DynamoRIO/dynamorio</a>	1.8k	Google 出品的跨平台动态二进制插桩框架。
<a href="#">intel/pin</a>	N/A	Intel 出品的动态二进制插桩框架。
<a href="#">googleprojectzero/winafl</a>	2.1k	AFL 的一个分支，用于对 Windows 二进制文件进行模糊测试。
<a href="#">processhacker/processhacker</a>	3.2k	强大的多用途工具，用于监控系统资源、调试软件和检测恶意软件。
<a href="#">dsincli/procmon-parser</a>	170	Sysinternals Process Monitor (Procmon) 的日志解析器。
<a href="#">microsoft/Detours</a>	1.9k	微软官方的 API Hooking 工具库。
<a href="#">easyhook/EasyHook</a>	1.8k	强大的 Windows API Hooking 库。
<a href="#">tmate-io/tmate</a>	3.5k	即时终端共享工具。
<a href="#">lief-project/LIEF</a>	4k	用于解析、修改和抽象 ELF, PE, MachO 格式的库。
<a href="#">qbdi/QBDI</a>	680	基于 LLVM 的动态二进制插桩框架。
<a href="#">jmpewws/dobby</a>	1.6k	轻量级、多平台、多架构的 Hook 框架。
<a href="#">aslody/whale</a>	880	跨平台的 Hook 框架 (Android/iOS/Linux/macOS)。
<a href="#">iqiyi/xHook</a>	1.6k	用于 Android aarch64/arm/x86 平台的 PLT hook 库。

项目	Star 数量	描述
<a href="#">facebook/fishhook</a>	3.8k	在 iOS/macOS 上动态重绑定 Mach-O 二进制文件中的符号。

## 2. 反汇编器与反编译器

项目	Star 数量	描述
<a href="#">NationalSecurityAgency/ghidra</a>	47.9k	NSA 出品的软件逆向工程框架，包含反编译器。
<a href="#">radareorg/radare2</a>	19.8k	开源的逆向工程框架和命令行工具集。
<a href="#">rizin-re/rizin</a>	2.5k	radare2 的一个分支，专注于可用性和社区。
<a href="#">avast/retdec</a>	8.2k	基于 LLVM 的可重定向机器码反编译器。
<a href="#">yegord/snowman</a>	1.8k	支持 x86, ARM 和 x86-64 的反编译器。
<a href="#">aquynh/capstone</a>	7.4k	强大的多架构反汇编框架。
<a href="#">keystone-engine/keystone</a>	3.9k	轻量级多架构汇编器框架。
<a href="#">unicorn-engine/unicorn</a>	7.5k	基于 QEMU 的多架构 CPU 模拟器框架。
<a href="#">lifting-bits/mcsema</a>	1.7k	将 x86/64, aarch64 二进制文件提升到 LLVM IR。
<a href="#">maji-cat/redress</a>	680	基于 Triton 的二进制反编译器。
<a href="#">wtdcode/retdec-idapluging</a>	500+	RetDec 反编译器的 IDA 插件。
<a href="#">airbus-seclab/bincat</a>	1k	二进制代码静态分析工具，支持值分析、污点分析和类型推断。

### 3. 调试器

项目	Star 数量	描述
<a href="#">x64dbg/x64dbg</a>	45.3k	Windows 平台开源的 x64/x32 调试器。
<a href="#">gdb/gdb</a>	N/A	GNU 项目调试器。
<a href="#">radareorg/cutter</a>	15.6k	radare2 的 GUI 界面。
<a href="#">hugsy/gef</a>	6.4k	GDB 的现代化插件，用于漏洞利用和逆向。
<a href="#">pwndbg/pwndbg</a>	6.7k	GDB 的一个插件，辅助 pwn。
<a href="#">longld/peda</a>	5.7k	GDB PEDA - Python Exploit Development Assistance for GDB.
<a href="#">voltron/voltron</a>	5.3k	一个可扩展的、跨平台的调试器 UI 工具包。
<a href="#">microsoft/WinDbg-Samples</a>	300+	WinDbg 的示例扩展、脚本和 API 用法。
<a href="#">moyix/pdbpp</a>	1.4k	Python 调试器 (pdb) 的一个增强版。
<a href="#">deroko/x64dbg-python</a>	300+	用于 x64dbg 的 Python 脚本插件。

## 4. 静态分析与二进制分析

项目	Star 数量	描述
<a href="#">angr/angr</a>	7.3k	强大的二进制分析平台，支持符号执行。
<a href="#">trailofbits/manticore</a>	2k	动态二进制分析工具，支持符号执行、污点分析。
<a href="#">JonathanSalwan/triton</a>	2.7k	动态二进制分析 (DBA) 框架。
<a href="#">google/binexport</a>	450+	将反汇编从 IDA Pro, Binary Ninja, Ghidra 导出到 BinNavi。
<a href="#">google/binnavi</a>	2.8k	二进制代码逆向工程和分析的图形化工具。
<a href="#">Gallopsled/pwnools</a>	11.2k	CTF 框架和漏洞利用开发库。
<a href="#">erocarrera/pefile</a>	1.3k	用于解析和操作 PE 文件的 Python 模块。
<a href="#">eliben/pyelftools</a>	1k	用于解析和分析 ELF 文件和 DWARF 调试信息的 Python 库。
<a href="#">s-c-repo/vtable-dumper</a>	250	用于从 PE/ELF 文件中 dump 虚函数表的工具。

## 5. android 平台

项目	Star 数量	描述
<a href="#">iBotPeaches/Apktool</a>	18.2k	用于逆向 Android apk 文件的工具。
<a href="#">pxb1988/dex2jar</a>	12k	用于处理 .dex 和 .class 文件的工具。
<a href="#">skylot/jadx</a>	38.6k	Dex 到 Java 的反编译器。
<a href="#">JesusFreke/smali</a>	4.4k	Android 的 smali/baksmali 汇编器/反汇编器。
<a href="#">MobSF/Mobile-Security-Framework-MobSF</a>	16.4k	自动化的移动应用 (Android/iOS/Windows) 安全测试和恶意软件分析框架。
<a href="#">sensepost/objection</a>	8.3k	运行时移动安全评估框架，基于 Frida。
<a href="#">Fuzion24/JustTrustMe</a>	2.2k	禁用 SSL 证书检查的 Xposed 模块。
<a href="#">ac-pm/Inspeckage</a>	1.8k	Android 包动态分析工具，带 API hook 功能。
<a href="#">rednaga/APKiD</a>	850	用于识别 Android 安装包中加壳、混淆和其它异常的工具。
<a href="#">CalebFenton/simplify</a>	3.2k	通用 Android 反混淆工具。
<a href="#">strazzere/android-unpacker</a>	900+	Defcon 22 上演示的 Android 脱壳工具。
<a href="#">asLody/AndHook</a>	600+	Android 动态插桩框架。
<a href="#">turing-technician/fasthook</a>	400+	Android ART Hook 框架。
<a href="#">wrbbug/dumpdex</a>	1.9k	Android 脱壳工具。
<a href="#">topjohnwu/Magisk</a>	35k+	Android 系统无感知 Root 工具。
<a href="#">LSPosed/LSPosed</a>	15k+	

项目	Star 数量	描述
		基于 Riru/Zygisk 的 ART Hook 框架 (Xposed 替代品)。
<a href="#">Genymobile/scrcpy</a>	90k+	高性能的 Android 投屏与控制工具。
<a href="#">shroudedcode/apk-mitm</a>	3k+	自动修改 APK 以便进行 HTTPS 抓包的工具。
<a href="#">rOysue/r0capture</a>	4k+	基于 Frida 的安卓应用层抓包通杀脚本。

## 6. 多平台与通用工具

项目	Star 数量	描述
<a href="#">upx/upx</a>	4.9k	极致的可执行文件压缩器。
<a href="#">horsicq/Detect-It-Easy</a>	5.8k	用于判断文件类型的程序，支持 Windows, Linux, macOS。
<a href="#">wbenny/xorsearch</a>	450+	在文件中搜索经过 XOR, ROL, ROT 或 SHIFT 编码的字符串。
<a href="#">balena-io/etason</a>	200+	JSON 解析器，能容忍语法错误。
<a href="#">google/santa</a>	3.1k	用于 macOS 的二进制文件白名单/黑名单系统。
<a href="#">trailofbits/osquery-extensions</a>	300+	osquery 的扩展，用于增强安全分析。
<a href="#">checkra1n/pongoOS</a>	1.2k	checkra1n 使用的 Pre-boot eXecution Environment。
<a href="#">mitmproxy/mitmproxy</a>	33k+	交互式的 HTTPS 代理，用于调试、测试和渗透。

## 7. Hex 编辑器

项目	Star 数量	描述
<a href="#">codepainters/distorm</a>	500+	x86/AMD64 的快速反汇编库。
<a href="#">radareorg/hex-lib</a>	100+	Hex a go go.
<a href="#">WerWolv/ImHex</a>	3k	一个功能丰富的现代 Hex 编辑器。
<a href="#">chrystianvieira/hexcute</a>	100+	一个简单的十六进制编辑器。

## 8. 脱壳与反混淆

项目	Star 数量	描述
<a href="#">de4dot/de4dot</a>	6.5k	.NET 反混淆器和脱壳器。
<a href="#">fireeye/flare-floss</a>	1.5k	自动从恶意软件中提取混淆后的字符串。
<a href="#">ioncodes/dnpatch</a>	500+	用于修补 .NET 程序集的工具。
<a href="#">rolfrolles/deobfuscator</a>	400+	基于 QEMU 的 x86 反混淆器。
<a href="#">hluwa/frida-dexdump</a>	3k+	基于 Frida 的快速 Dex 内存导出工具。
<a href="#">Perfare/Il2CppDumper</a>	7k+	Unity Il2Cpp 逆向工具，还原 DLL 和头文件。

## 9. 固件分析

项目	Star 数量	描述
<a href="#">ReFirmLabs/binwalk</a>	10.1k	用于分析、逆向和提取固件镜像的工具。
<a href="#">craigz/firmwalker</a>	1k	自动在固件中搜索敏感信息的脚本。
<a href="#">attify/firmware-analysis-toolkit</a>	1k	用于固件安全测试的工具包。
<a href="#">scriptingx/IoTSecurity101</a>	500+	物联网安全入门指南。

## 10. Apple 平台 (iOS/macOS)

项目	Star 数量	描述
<a href="#">nygard/class-dump</a>	2.6k	从 Mach-O 文件生成 Objective-C 头文件。
<a href="#">KJCracks/Clutch</a>	2.8k	快速的 iOS 可执行文件 dumper。
<a href="#">alonemonkey/MonkeyDev</a>	4.5k	iOS Tweak 开发工具，无需越狱。
<a href="#">facebook/chisel</a>	8.3k	辅助调试 iOS 应用的 LLDB 命令集合。
<a href="#">nabla-c0d3/ssl-kill-switch2</a>	1.6k	黑盒工具，用于在 iOS 和 macOS 应用中禁用 SSL 证书验证。
<a href="#">ptoomey3/Keychain-Dumper</a>	1k	在越狱设备上检查哪些钥匙串项可被访问。
<a href="#">limneos/classdump-dyld</a>	450+	无需从 dyld_shared_cache 中提取即可 class-dump 任何 Mach-O 文件。

## 11. 其他与资源

项目	Star 数量	描述
<a href="#">firmianay/security-paper</a>	1.7k	安全领域的一些经典论文。
<a href="#">endgameinc/RSOI</a>	200+	逆向工程领域的资源和信息。
<a href="#">enaqx/awesome-pentest</a>	18k+	精选的渗透测试资源、工具和其它很棒的东西。
<a href="#">carpedm20/awesome-hacking</a>	9k+	精选的黑客资源、工具和教程。
<a href="#">onethawt/idaplugins-list</a>	1.9k	IDA Pro 插件列表。
<a href="#">Siguza/ios-resources</a>	700+	iOS 黑客相关的有用资源。
<a href="#">michalmalik/osx-re-101</a>	1.4k	OSX/iOS 逆向资源。

## [X02] 学习资源

## X02: 学习资源 (Learning Resources)

本页面收集了 Android 逆向工程领域的高质量学习资源，包括书籍、博客、论坛、社区和课程。

### 书籍 (Books)

#### 入门与基础

- 《Android 软件安全与逆向分析》 (丰生强 / 非虫) 经典的入门书籍，涵盖了 Android 系统架构、Smali 语法、静态分析、动态调试等基础知识。
- 《Android 安全攻防实战》 (EaaLaboratory) 倾向实战，包含很多案例分析。

#### 进阶与深入

- 《Android Internals: A Confectioner's Cookbook》 (Jonathan Levin) [链接](#) 深入剖析 Android 系统内部原理，是理解 Android 底层机制的必读之作。
- 《Android Hacker's Handbook》 (Joshua J. Drake et al.) 全面介绍 Android 安全架构、漏洞挖掘和利用技术。
- 《MASTG - Mobile App Security Testing Guide》 (OWASP) [链接](#) OWASP 发布的移动应用安全测试指南，涵盖了 iOS 和 Android 平台的安全测试方法论和技术细节，是行业标准参考文档。

## 博客与网站 (Blogs & Websites)

### 个人博客

- Maddie Stone ([Project Zero](#)) 专注于 Android 恶意软件分析和漏洞挖掘，文章质量极高。
- ROysue (肉丝) 国内知名的 Android 逆向专家，Frida 领域的领军人物。
- Wei (LSPosed Developer) 深入研究 Android Runtime (ART) 和 Hook 技术。
- Orange Tsai Web 和移动安全领域的知名研究员，常有精彩的利用思路。

### 技术团队与厂商

- Google Project Zero [链接](#) Google 的安全研究团队，发布了大量关于 Android 内核、驱动和框架层的高质量漏洞分析报告。
- QuarksLab Blog [链接](#) 发布了许多关于混淆、反混淆和底层逆向工具（如 Triton）的研究文章。
- Check Point Research 经常披露 Android 恶意软件和高危漏洞。

## 论坛与社区 (Forums & Communities)

- 52pojie (吾爱破解) [链接](#) 国内最大的破解和逆向技术交流论坛，拥有丰富的教程、工具和活跃的社区氛围。
- Kanxue (看雪论坛) [链接](#) 国内老牌的安全技术社区，专注于二进制安全、漏洞挖掘和内核安全，技术深度较高。
- XDA Developers [链接](#) 全球最大的 Android 开发者社区，关于 ROM 定制、Root、Xposed/Magisk 模块的资源非常丰富。
- Reddit r/ReverseEngineering [链接](#) 国际逆向工程技术讨论区，汇集了全球的逆向爱好者和专家。

- 
- Reddit r/androiddev [链接](#) 虽然侧重开发，但了解开发者的思维对于逆向工程也非常有帮助。
- 

## 课程与教程 (Courses & Tutorials)

- Frida 官方文档与教程 [链接](#) 学习 Frida 最权威的资料。
  - Android App Reverse Engineering 101 (Maddie Stone) [链接](#) (需查找有效链接或存档) Workshops 形式的入门教程，非常适合初学者。
  - OWASP Mobile Security Testing Guide (MSTG) Hacking Playground 配合 MSTG 书籍的练习环境。
- 

## 其他资源

- Android Open Source Project (AOSP) [链接](#) 阅读源码是理解 Android 最根本的方法。使用 [cs.android.com](http://cs.android.com) 进行在线源码搜索非常方便。
  - Android Developers Documentation [链接](#) 官方开发文档，逆向时遇到不懂的 API 首先应该查阅的地方。
-

## [X03] CTF 练习平台

## X03: CTF 与练习平台

---

实践是掌握逆向工程技术的关键。本页面整理了提供 Android 逆向挑战的 CTF 平台和 CrackMe 网站。

### 移动安全专项挑战 (Mobile Specific Challenges)

#### OWASP UnCrackable Apps

- 描述: OWASP 官方提供的一系列 Android 和 iOS 逆向挑战应用，分为 Level 1 到 Level 4 不同难度。是学习移动安全测试标准 (MSTG) 的最佳配套练习。
- 链接: [OWASP MSTG Repo](#)

#### Google CTF (Mobile Category)

- 描述: Google 每年举办的 CTF 比赛中的 Mobile 类目题目。这些题目通常质量很高，涉及各种 Android 特性和新颖的保护机制。
- 链接: [Google CTF Archives](#) (查看历年题目)

#### android App Reverse Engineering 101 Crackmes

- 描述: Maddie Stone 在她的 Workshop 中使用的练习题目。
  - 链接: [GitHub Repo](#)
-

## 综合 CTF 平台 (General CTF Platforms)

### Hack The Box (HTB)

- 描述: 著名的渗透测试练习平台, 其中也有不少 Android 逆向相关的 Challenge (通常在 Mobile 或 Reversing 分类下) 和 Machine。
- 链接: <https://www.hackthebox.com/>

### TryHackMe

- 描述: 对初学者更友好的网络安全学习平台, 提供有引导性的 Android 逆向房间 (Rooms)。
- 链接: <https://tryhackme.com/>

### CTFtime

- 描述: 全球 CTF 赛事聚合平台。可以在这里关注即将开始的比赛, 很多综合性比赛都会包含 Reverse 和 Mobile 方向的题目。
- 链接: <https://ctftime.org/>

## CrackMe 网站

### Crackmes.one

- 描述: 全球最大的 CrackMe 收集网站。你可以通过搜索 "Android" 或 "APK" 标签找到大量的 Android 逆向练习程序, 难度从简单到极难都有。
- 链接: <https://crackmes.one/>

---

## Root Me

- 描述: 一个涵盖各种安全领域的练习平台, 其 "Cracking" 和 "App - Script" 分类下有一些针对移动应用的挑战。
  - 链接: <https://www.root-me.org/>
- 

## 推荐练习路线

1. 入门: 从 OWASP UnCrackable Level 1 开始, 学习基本的反编译、代码分析和简单的逻辑绕过。
  2. 进阶: 尝试 Crackmes.one 上评分较高的 Android 题目, 或者 Hack The Box 的简单 Mobile 挑战。
  3. 高级: 挑战 Google CTF 的历史题目, 或者 OWASP UnCrackable Level 3/4, 主要涉及各种反调试、Native 层混淆、壳分析等。
-

---

## [X04] 术语表

# X04: 术语表 (Glossary)

收集了 Android 逆向工程中常见的术语和缩写。

## A

- ADB (Android Debug Bridge): Android 调试桥，一个通用的命令行工具，允许你与模拟器实例或连接的 Android 设备进行通信。
- AOSP (Android Open Source Project): Android 开源项目，即 Android 系统的源代码。
- APK (Android Package): Android 应用程序包，Android 操作系统使用的一种应用程序包文件格式。
- ART (Android Runtime): Android 运行时，Android 5.0 引入的新的应用运行时环境，完全取代了 Dalvik。它使用 AOT (Ahead-Of-Time) 编译技术。
- ARM: 一种精简指令集 (RISC) 处理器架构，广泛用于移动设备。

## B

- Bootloader: 引导加载程序，在操作系统内核运行之前运行的一段小程序，负责加载操作系统。
- Baksmali: 一个将 dex 文件反汇编成 smali 文件的工具。

## D

- Dalvik: Google 早期为 Android 设计的虚拟机，使用 JIT (Just-In-Time) 编译。在 Android 5.0 后被 ART 取代。
- DEX (Dalvik Executable): Android 平台的可执行文件格式，包含编译后的代码。

- Dynamic Analysis (动态分析): 在程序运行时对其进行分析的技术，通常涉及调试、Hook 等。

## E

- ELF (Executable and Linkable Format): 可执行与可链接格式，Linux 系统（包括 Android Native 层）使用的标准二进制文件格式。

## F

- Frida: 一个动态插桩工具包，允许开发者、逆向工程师和安全研究人员在运行时监视和修改应用程序的行为。

## G

- Ghidra: NSA 开源的软件逆向工程 (SRE) 框架。

## H

- Hooking (挂钩): 一种拦截软件组件之间函数调用、消息或事件的技术，用于改变或监视系统的行为。

## I

- IDA Pro (Interactive DisAssembler): 业界标准的交互式反汇编器和调试器。
- IL2CPP: Unity 游戏引擎的一种脚本后端，将 C# 代码转换为 C++ 代码，增加了逆向难度。

## J

- JAD: 一个将 DEX 文件反编译为 Java 代码的工具。
- JNI (Java Native Interface): Java 本地接口, 允许 Java 代码和其他语言 (主要是 C/C+++) 写的代码进行交互。

## M

- Magisk: 一个开源的 Android Root 解决方案, 以 "Systemless" (不修改系统分区) 著称。
- Manifest (AndroidManifest.xml): 每个 Android 应用都必须包含的文件, 描述了应用的包名、组件、权限等基本信息。

## N

- Native Code: 通常指使用 C/C++ 编写的, 直接编译为机器码的代码 (相对于 Java/Kotlin 字节码)。
- NDK (Native Development Kit): 一个工具集, 允许开发者使用 C 和 C++ 实现应用的一部分。

## O

- Obfuscation (混淆): 使代码难以理解但保持其功能不变的技术, 用于保护知识产权或隐藏恶意行为。
- OLLVM (Obfuscator-LLVM): 基于 LLVM 的代码混淆项目, 常用于 Native 代码混淆。
- OAT: ART 运行时使用的私有 ELF 文件格式, 包含 AOT 编译后的机器码。

## R

- Recovery: Android 设备的恢复模式, 用于恢复出厂设置、刷入更新包等。

- 
- Rooting: 获取 Android 设备超级用户 (Root) 权限的过程。
  - Riru: 一个用于注入 Zygote 进程的模块，常作为其他模块（如 LSposed）的基础。

## S

- Smali: Android 的 Dalvik 字节码的人类可读汇编语言。
- Static Analysis (静态分析): 在不运行程序的情况下对其进行分析的技术。
- So (Shared Object): Linux/Android 下的动态链接库文件，通常由 C/C++ 编写。

## V

- VMP (Virtual Machine Protection): 虚拟机保护，一种高级混淆技术，将原始代码转换为自定义字节码并在自定义解释器中运行。

## X

- Xposed: 一个强大的 Android 框架，允许在不修改 APK 的情况下通过模块改变系统和应用的行为。

## Z

- Zygote: Android 系统中所有应用进程的父进程。

## 配方编号系统

# 配方编号系统

本书采用系统化的编号方式组织内容，帮助你快速定位所需信息。

## 章节编号规则

本书分为六大部分，每部分有独立的编号前缀：

编号前缀	章节名称	内容定位	示例
R	Recipes	场景化解决方案（核心内容）	R01, R02...
T	Tools	工具使用指南与原理	T01, T02...
C	Case Studies	实战案例分析	C01, C02...
F	Foundations	基础知识参考	F01, F02...
A	Advanced	高级主题参考	A01, A02...
E	Engineering	工程化参考	E01, E02...
X	Appendix	附录资源与索引	X01, X02...

## Recipes 分类编号

Recipes（配方）是本书的核心内容，按照问题场景分类：

### 01-Recipes/Network (NET)

网络与加密相关配方

配方编号	配方名称	解决的问题
NET-01	Network Sniffing	网络抓包与协议分析
NET-02	Crypto Analysis	加密算法分析与还原
NET-03	TLS Fingerprinting	TLS 指纹识别与处理
NET-04	JA3 Fingerprinting	JA3 指纹分析
NET-05	JA4 Fingerprinting	JA4 指纹分析

### 01-Recipes/Anti-Detection (AD)

反检测与对抗相关配方

配方编号	配方名称	解决的问题
AD-01	Frida Anti Debugging	Frida 反调试检测绕过
AD-02	Xposed Anti Debugging	Xposed 反检测绕过
AD-03	Captcha Bypassing	验证码识别与绕过
AD-04	App Hardening ID	加固方案识别
AD-05	Device Fingerprinting	设备指纹采集与绕过
AD-06	Mobile App Security	移动端安全与反机器人

## 01-Recipes/Unpacking (UNP)

脱壳与修复相关配方

配方编号	配方名称	解决的问题
UNP-01	Unpacking	通用脱壳技术
UNP-02	Frida Unpacking	Frida 脱壳与 SO 修复
UNP-03	SO Deobfuscation	SO 文件反混淆
UNP-04	SO String Deobfuscation	SO 字符串解密

## 01-Recipes/Analysis (ANA)

分析与调试相关配方

配方编号	配方名称	解决的问题
ANA-01	RE Workflow	逆向工程标准工作流
ANA-02	Static Analysis	静态分析深入
ANA-03	Dynamic Analysis	动态分析深入
ANA-04	OLLLVM Deobfuscation	OLLLVM 混淆还原
ANA-05	VMP Analysis	虚拟机保护分析
ANA-06	JS Obfuscator	JavaScript 混淆分析
ANA-07	JS VMP	JavaScript 虚拟机保护
ANA-08	Native String	Native 字符串混淆

## 01-Recipes/Automation (AUTO)

自动化与规模化相关配方

配方编号	配方名称	解决的问题
AUTO-01	Device Farming	自动化与设备群控
AUTO-02	Dial Up Proxy	动态拨号代理池
AUTO-03	Proxy Pool Design	代理池架构设计
AUTO-04	Scrapy	Scrapy 爬虫框架
AUTO-05	Scrapy Redis	分布式 Scrapy
AUTO-06	Docker Deployment	Docker 容器部署
AUTO-07	Virtualization	虚拟化与容器技术
AUTO-08	Web Anti Scraping	Web 反爬虫对抗

## 01-Recipes/Scripts (SCR)

### 脚本集合

配方编号	配方名称	内容说明
SCR-01	Frida Examples	Frida 脚本示例集
SCR-02	Frida Common	Frida 常用脚本
SCR-03	Automation Scripts	自动化脚本集
SCR-04	Native Hooking	Native Hook 模式
SCR-05	Objection Snippets	Objection 代码片段
SCR-06	C For Emulation	C 语言仿真代码

## Tools 编号 (T)

工具使用指南和原理剖析，按功能分类。

### Dynamic - 动态分析工具

编号	工具名称	内容类型
T01	Frida Guide	使用指南
T02	Frida Internals	内部原理
T03	Xposed Guide	使用指南
T04	Xposed Internals	内部原理
T05	Unidbg Guide	使用指南
T06	Unidbg Internals	内部原理

### Static - 静态分析工具

编号	工具名称	内容类型
T07	Ghidra Guide	使用指南
T08	IDA Pro Guide	使用指南
T09	Radare2 Guide	使用指南

## Cheatsheets - 速查表

编号	名称	内容说明
T10	ADB Cheatsheet	ADB 命令速查

## Case Studies 编号 (C)

真实场景的案例分析，综合运用各类技术。

编号	案例名称	分析对象
C01	Anti Analysis Techniques	反分析技术案例
C02	Music Apps	音乐 App 分析
C03	Social Media & Anti Bot	社交媒体与风控
C04	App Encryption	应用加密案例
C05	Video Apps & DRM	视频 App 与 DRM
C06	Unity Games (Il2Cpp)	Unity 游戏分析
C07	Flutter Apps	Flutter 应用分析
C08	Malware Analysis	恶意软件分析

## Reference 编号

参考资料和理论知识，按主题分类。

## Foundations 基础知识 (F)

编号	主题	内容说明
F01	APK Structure	APK 文件结构
F02	Android Components	四大组件
F03	Android Manifest	清单文件解析
F04	Android Studio Debug Tools	调试工具
F05	DEX Format	DEX 文件格式
F06	Smali Syntax	Smali 语法
F07	SO ELF Format	ELF 文件格式
F08	ART Runtime	ART 运行时
F09	ARM Assembly	ARM 汇编
F10	x86 & ARM Assembly Basics	汇编基础

## Advanced 高级主题 (A)

编号	主题	内容说明
A01	Android Sandbox Implementation	沙箱实现原理
A02	AOSP & System Customization	AOSP 系统定制
A03	AOSP Device Modification	设备修改
A04	Minimal Android Rootfs	最小化根文件系统
A05	SO Anti Debugging & Obfuscation	SO 反调试与混淆
A06	SO Runtime Emulation	SO 运行时仿真

## Engineering 工程化 (E)

编号	主题	内容说明
E01	Frameworks & Middleware	框架与中间件
E02	Message Queues	消息队列
E03	Redis	Redis 数据库
E04	Risk Control SDK Build Guide	风控 SDK 构建
E05	Data Warehousing & Processing	数据仓库与处理
E06	Flink	流处理框架
E07	HBase	分布式数据库
E08	Hive	数据仓库
E09	Spark	大数据处理

## Appendix 编号 (X)

附录资源和社区资源。

编号	名称	内容说明
X01	Github Projects	开源项目推荐
X02	Learning Resources	学习资源汇总
X03	CTF Platforms	CTF 平台推荐
X04	Glossary	术语表

## 如何使用编号系统

### 快速引用

在讨论或笔记中，你可以使用简短的编号引用特定配方：

- "参考 R02 了解加密分析方法"
- "这个问题可以用 R06 的方法解决"
- "代码示例见 R32"
- "工具使用参考 T01"

### 交叉引用

本书各章节之间存在大量交叉引用，编号系统帮助你快速定位：

分析加密算法时（R02），可能需要先脱壳（R12），  
然后使用 Frida（T01）进行动态分析（R18），  
参考 C04 了解完整案例。

### 学习路径规划

你可以根据编号规划自己的学习路径：

---

初学者路径：

F01 → F05 → T01 → R01 → R06

爬虫开发者路径：

R01 → R02 → R10 → R27 → R28

安全研究员路径：

R16 → R12 → R19 → A05 → C08

---

## 版本说明

本编号系统会随着内容更新而扩展。新增配方将获得新的编号，已有配方的编号保持不变，确保引用的稳定性。

当前版本：v1.0

---

现在你已经了解了本书的组织结构，开始你的学习之旅吧！

推荐下一步：[10 分钟快速入门](#)