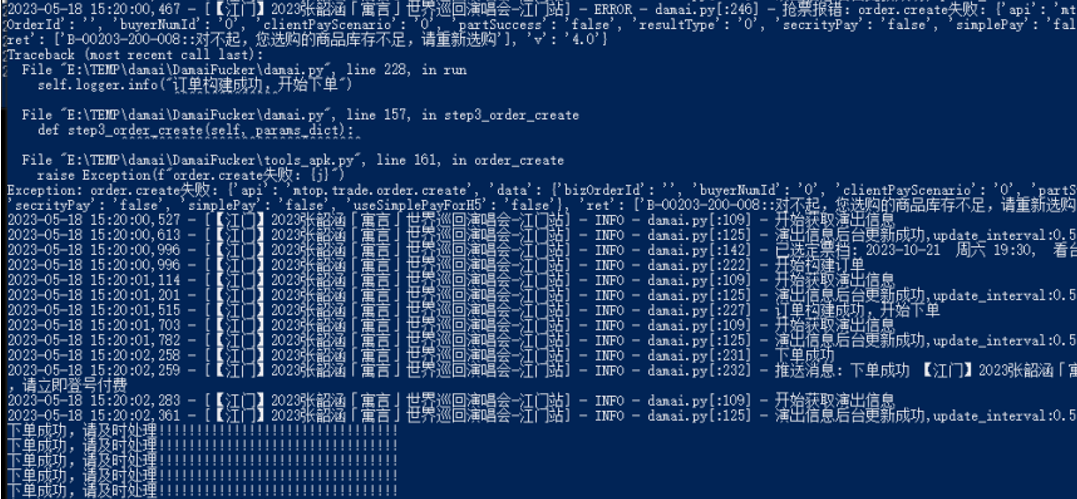
**概述**

本文以初学者的身份，记录从0开始学习逆向相关内容并实现某购票APP的自动化抢票的过程，详细介绍需要用到的工具，先验知识，以及分析过程。文章顺序为：环境搭建、抓包、trace分析、接口参数获取、RPC调用实现，以及最终的功能实现。

本文借鉴了各大论坛中的相关经验，会在文中给出一些链接

效果图：



**一、缘起**

在疫情放开之后，大家的出游频率和规模都出现了大幅增长，演唱会也多了起来，但是场次虽然多，票却抢不到。身为计算机从业者，有点受不了这个委屈（开玩笑），于是想着干脆就借着这个机会从头开始学习逆向，自己做一个自动化购票的东西。

**二、环境**

* windows 10
* cn.damai apk 版本8.5.4 (2023-04-26)
* bluestacks 5.11.56.1003 p64
* adb 31.0.2
* Root Checker 6.5.3
* wireshark 4.0.5
* frida 16.0.19
* jadx-gui 1.4.7

**三、环境搭建**

**bluestacks 环境搭建**

**首先需要root Bluestacks环境。**

1. 下载安装Bluestacks。
2. 运行Bluestacks Multi-instance Manager，发现默认安装的版本为Android Pie 64bit版本，即Android 9.0。此时退出bluestack所有程序。
3. 关闭bluestack后编辑bluestacks配置文件， %programdata%\BlueStacks\_nxt\bluestacks.conf
4. 在配置文件中查找root关键词，对应值修改为1，共两处。

bst.feature.rooting="1"  
bst.instance.Pie64.enable\_root\_access="1"

1. 启动bluestack模拟器，安装 Root Checker APP，点击验证root，即可发现root已成功。

上述 root 过程主要参考了 https://appuals.com/root-bluestacks/ ，部分地方做了改正

**打开 adb调试**

1. bluestack设置-高级中打开Adb调试，并记录下端口
2. 打开主机命令行，运行 adb connect localhost:6652，端口号修改为上一步的端口号，即可连接。再运行adb devices，如有对应设备则连接成功。
3. 进入adb shell，执行su进入root权限，命令行标识由$变为#，即表示adb 进入root权限成功。

**frida环境搭建**

frida可以直接访问修改二进制的内存、函数和对象，对于Android的支持也很理想。

frida的运行采用C/S架构，客户端为电脑端的开发环境，服务器端为Android，均需对应部署搭建。

**客户端环境搭建(Windows)**

firda客户端基于python3开发，因此首先需要配置好python3的运行环境，然后执行 pip install frida-tools即可完成安装。运行 frida --version可验证frida版本。

(py3) PS E:\TEMP\damai> frida --version  
16.0.19

**服务器 环境搭建(Android)**

环境搭建第二步是在Android模拟器中运行frida-server。这样可以让Frida通过ADB/USB调试与Android模拟器连接。

1. 下载frida-server  
   最新的frida-server可以从 https://github.com/frida/frida/releases 下载。注意下载与设备匹配的架构。如果模拟器是x86\_64，则下载相应版本的frida-server。本文使用的版本为 frida-server-16.0.18-android-x86\_64.xz
2. 传入Android模拟器。

将下载后的.xz文件解压，将frida-server传入Android模拟器

adb push frida-server /data/local/tmp/

1. 运行 frida-server

使用adb root以root模式重新启动ADB，并通过adb shell重新进入shell的访问。进入shell后，进入放置frida-server的目录并为其授予执行权限：

cd /data/local/tmp/chmod +x frida-server

执行：./frida-server ，运行frida-server，并保持本shell窗口开启。

有些情况下，应用程序会检测在是否在模拟器中运行，但对该app的分析暂无影响。

1. 测试是否连接成功

在window端运行frida-ps命令：

如果可以看到一堆熟悉的Android进程，则表示连接成功

1. 转发frida-server端口 (可选)

frida-server跑在Android端，frida需要通过连接frida-server。上一步使用adb的方式连接，frida认为是USB模式，需要-U命令。frida也支持依赖端口的远程连接模式，在某些场景下更加灵活。可以通过端口转发的方式实现此功能。

adb forward tcp:27042 tcp:27042adb forward tcp:27043 tcp:27043

27042是用于与frida-server通信的默认端口号,之后的每个端口对应每个注入的进程，检查27042端口可检测 Frida 是否存在。

本部分主要参考了 https://learnfrida.info/java/

**四、抓包**

**抓包及https解密方法**

本节将介绍用tcpdump+frida+wireshark实现Android的全流量抓包，能实现https解密。

惯用的Android抓包手段是用fiddler/burpsuite/mitmproxy搭建代理服务器，设置Android代理服务器并用中间人劫持的方式获取http协议流量的内容。如需对https流量解密，还需要在安卓上安装https根证书。Android9.0以后的版本对用户自定义根证书有了一些限制，抓包不再那么简单，但可以参考以下几篇文章：

* 从原理到实战，全面总结 Android HTTPS 抓包
* Android 高版本 HTTPS 抓包解决方案

上述的抓包方式只能抓到http协议层以上的流量，这次决定采用tcpdump+frida+wireshark实现Android的全流量抓包，实现https解密。

**1. 搞定tcpdump**

本文基于termux安装使用tcpdump。

首先安装termux apk。

打开termux运行：

* 挂载存储

termux-setup-storage  
## 会弹出授权框，点允许  
ls ~/storage/  
## 如果出现dcim, downloads等目录，即表示成功

* 安装tcpdump

pkg install root-repopkg install sudo tcpdump

* 运行抓包

sudo tcpdump -i any -s 0 -w ~/storage/downloads/capture.pcap

之后就可以把downloads目录下的抓包文件拷贝到电脑上，用wireshark打开做进一步分析。

**2. 解密https流量**

Wireshark解密https流量的方法和原理介绍有很多，可参考以下文章，本文不再赘述。

* https://unit42.paloaltonetworks.com/wireshark-tutorial-decrypting-https-traffic/
* https://zhuanlan.zhihu.com/p/36669377

wireshark解密技术的重点在于拿到客户端通信的密钥日志文件(ssl key log)

在Android中实现抓取ssl key log需要hook系统的SSL相关函数，可以用frida实现。

* 首先将下面的hook代码保存为sslkeyfilelog.js

// sslkeyfilelog.js  
function startTLSKeyLogger(SSL\_CTX\_new, SSL\_CTX\_set\_keylog\_callback) {  
 console.log("start----")  
 function keyLogger(ssl, line) {  
 console.log(new NativePointer(line).readCString());  
 }  
 const keyLogCallback = new NativeCallback(keyLogger, 'void', ['pointer', 'pointer']);  
  
 Interceptor.attach(SSL\_CTX\_new, {  
 onLeave: function(retval) {  
 const ssl = new NativePointer(retval);  
 const SSL\_CTX\_set\_keylog\_callbackFn = new NativeFunction(SSL\_CTX\_set\_keylog\_callback, 'void', ['pointer', 'pointer']);  
 SSL\_CTX\_set\_keylog\_callbackFn(ssl, keyLogCallback);  
 }  
 });  
}  
startTLSKeyLogger(  
 Module.findExportByName('libssl.so', 'SSL\_CTX\_new'),  
 Module.findExportByName('libssl.so', 'SSL\_CTX\_set\_keylog\_callback')  
)

* 然后用frida加载运行hook

frida -U -l .\sslkeyfilelog.js -f cn.damai

* 最后，抓包结束后将得到的key保存到sslkey.txt，格式是下面这样的，不要掺杂别的。

CLIENT\_RANDOM 557e6dc49faec93dddd41d8c55d3a0084c44031f14d66f68e3b7fb53d3f9586d 886de4677511305bfeaee5ffb072652cbfba626af1465d09dc1f29103fd947c997f6f28962189ee809944887413d8a20  
CLIENT\_RANDOM e66fb5d6735f0b803426fa88c3692e8b9a1f4dca37956187b22de11f1797e875 65a07797c144ecc86026a44bbc85b5c57873218ce5684dc22d4d4ee9b754eb1961a0789e2086601f5b0441c35d76c448

在运行Frida Hook获取sslkey的同时，运行tcpdump抓包。抓包中依次测试获取详情页、选择价位、提交订单等操作，并对应记录下执行操作的时间，方便后续分析。

抓包完成后，用wireshark打开tcpdump抓包获得的pcap文件，在wireshark首选项-protocols-TLS中，设置 (Pre)-Master-Secret log filename为上述sslkey.txt。

即可实现https流量的解密。

本部分主要参考了 https://www.52pojie.cn/thread-1405917-1-1.html

**流量分析**

在上述步骤中拿到了解密后的流量，就能对APP的流量做进一步分析了。

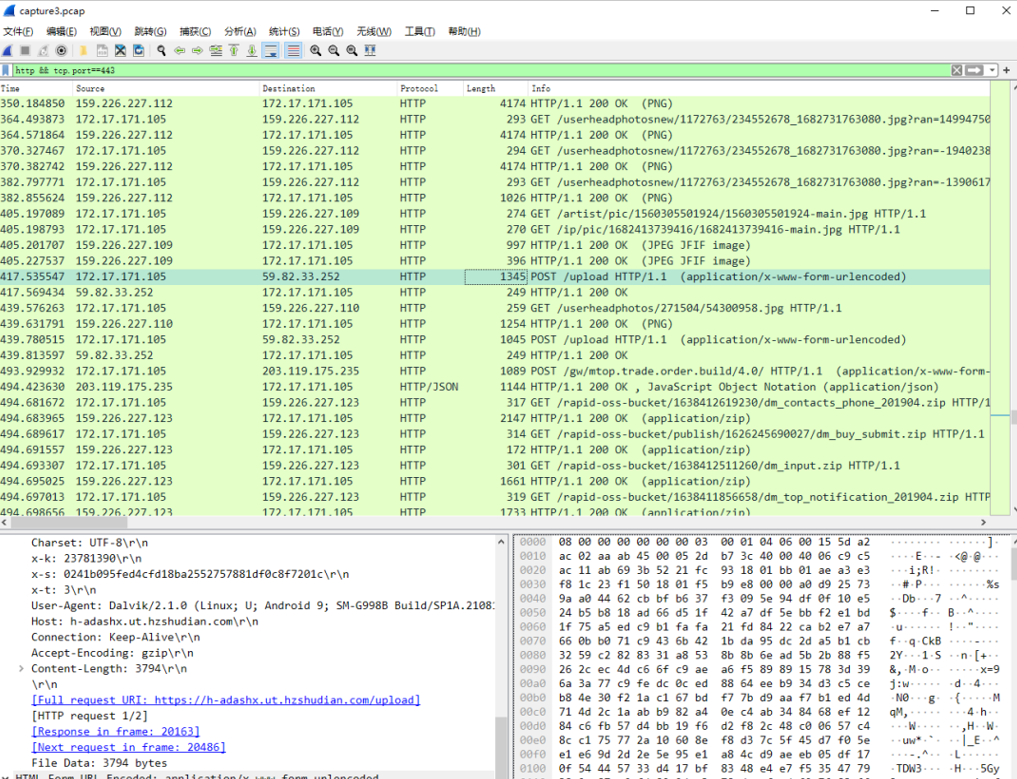
**APP的API流**

在此铺垫一下，通过前期对该APP的PC端和移动端H5的分析，购票的工作流程大概为：

1. 获得详情：接口为mtop.alibaba.damai.detail.getdetail。基于某演出的id(itemId)获得演出的详细信息，包括详情、场次、票档(SkuId)价位及状态信息，
2. 构建订单：接口为mtop.trade.order.build.h5。发送 演出id+数量+票档id(itemId\_count\_skuId)，得到提交订单所需的表单信息，包括观众、收货地址等。
3. 提交订单：接口为mtop.trade.order.create.h5。对上一步构建订单得到的表单参数作出修改后，发送给服务器，得到最后的订单提交结果和支付信息。

**apk流量分析**

首先用过滤器http && tcp.dstport==443，得到向服务器发送的https包，如下图：

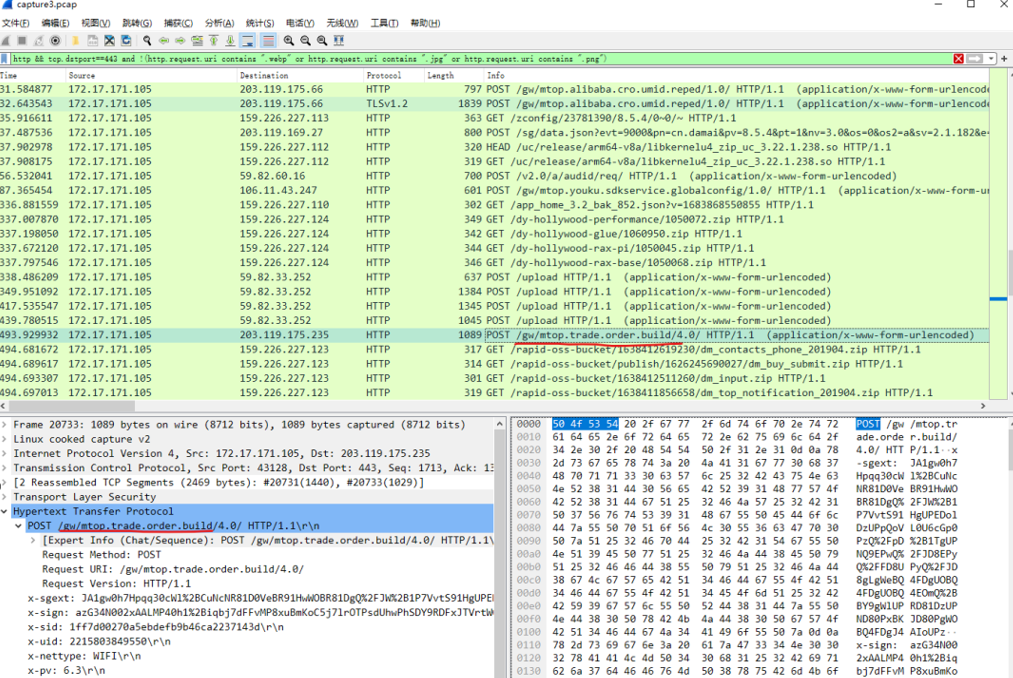


可以看到大量向服务器请求的数据包，但其中有很多干扰的图片请求，因为修改过滤器把图片过滤一下。过滤器：http && tcp.dstport==443 and !(http.request.uri contains ".webp" or http.request.uri contains ".jpg" or http.request.uri contains ".png")

结果会清晰很多。

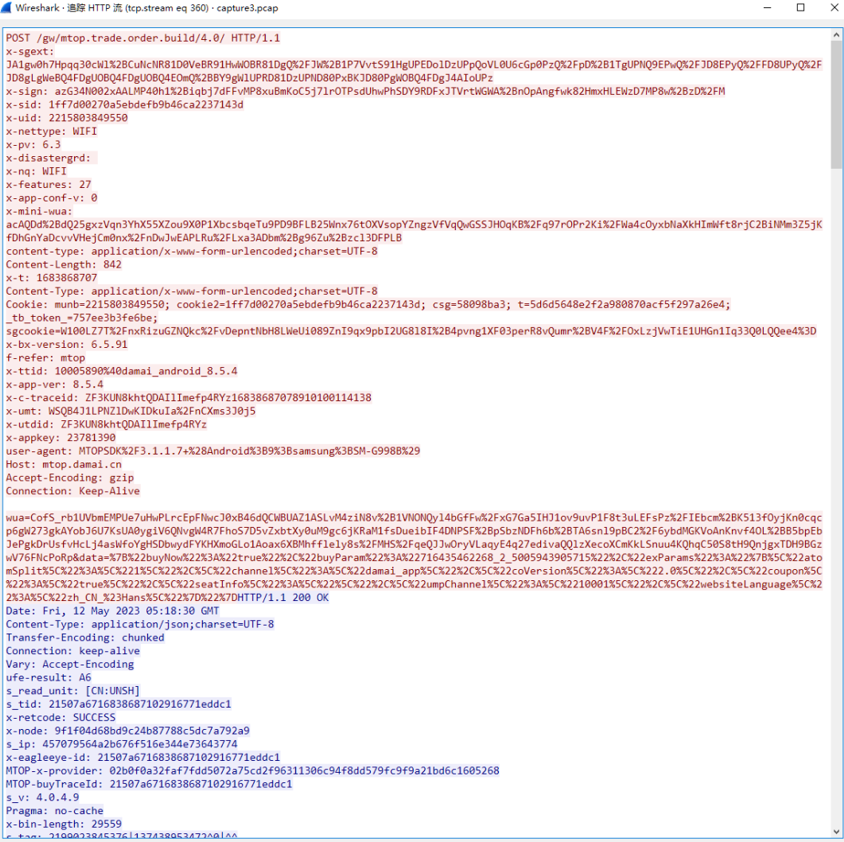
**订单构建(order.build)**

根据之前记录的操作的时间，以及对网页版的分析结果，注意到了下图的这条流量：



右键选择这条流量包，点击追踪http流，可以看到对应的响应包。





响应包里有些中文使用了UTF-8编码，可以点击右下角的Show data as，选择UTF-8，便可以正常显示。此时可以点击另存为，保存为txt文件，方便后续分析。

订单构建的请求包中核心的数据部分使用URL解码后为：

{"buyNow":"true","buyParam":"716435462268\_2\_5005943905715","exParams":"{\"atomSplit\":\"1\",\"channel\":\"damai\_app\",\"coVersion\":\"2.0\",\"coupon\":\"true\",\"seatInfo\":\"\",\"umpChannel\":\"10001\",\"websiteLanguage\":\"zh\_CN\_#Hans\"}"}

buyParam为最核心的部分，拼接方式为演出id+数量+票档id。其他部分只需照抄。

请求包中还包含大量的各种加密参数、ID，而破解实现自动购票脚本的关键就在于如何通过代码的方式拿到这些加密参数。

订单构建的响应包为订单提交表单的各项参数，用于生成“确认订单”的表单。



**订单提交(order.create)**

按照同样的方式可以找到订单提交包，订单提交包的API路径为/gw/mtop.trade.order.create，



其中青色圈出来的部分为data发送的核心数据，对数据用URL解码后为：

{"feature":"{\"gzip\":\"true\"}","params":"H4sIAAAAAAA.................AAWk3NKAAA\n"}

经过对朋友求助，他说看起来像是把原始数据用gzip压缩后又使用了base64编码，于是尝试解码：

import base64  
import gzip  
import json  
  
# 解码后变为python dict  
decode\_data=base64.b64decode(params\_str.replace("\\n",""))  
decompressed\_data=gzip.decompress(decode\_data).decode("utf-8")  
params=json.loads(decompressed\_data)  
  
with open("reverse\order.create-params.json","w") as f:  
 json.dump(params,f,indent=2)

解码成功，存到order.create-params.json,

解码后发现order.create发送的data参数和order.build请求返回的结果很相似，增加了一些用户对表单操作的记录。

order.create请求的header中的各种加密参数和order.build一致。

order.create请求的返回结果中包含了订单创建是否成功的结果以及支付链接。

**五、trace分析**

通过前面对流量的分析，已经知道客户端向服务器发送的核心数据和加密参数，核心数据的拼接相对简单，但加密参数怎么获得还比较困难。因此，下面要开始分析加密参数的生成方法。本节主要采用frida trace动态分析和jadx静态分析相结合的方式，旨在找到加密参数生成的核心函数和输入输出数据的格式。

根据文章 ( app安卓逆向x-sign，x-sgext，x\_mini\_wua，x\_umt加密参数解析 )，其中数据包的加密参数和本文的APP很类似，而且提到了 mtopsdk.security.InnerSignImpl 生成的加密函数，本文也参考了这篇文章的思路进行分析。

**跟踪 InnerSignImpl**

运行frida-trace -U -j "\*InnerSignImpl\*!\*" 大麦，执行选座提交订单的操作，发现确实有结果输出：

(py3) PS E:\TEMP\damai> frida-trace -U -j "\*InnerSignImpl\*!\*" 大麦  
Instrumenting...  
InnerSignImpl$1.$init: Loaded handler at "E:\\TEMP\\damai\\\_\_handlers\_\_\\mtopsdk.security.InnerSignImpl\_1\\\_init.js"  
....此处省略...  
InnerSignImpl.init: Loaded handler at "E:\\TEMP\\damai\\\_\_handlers\_\_\\mtopsdk.security.InnerSignImpl\\init.js"  
Started tracing 27 functions. Press Ctrl+C to stop.  
 /\* TID 0x144f \*/  
 6725 ms InnerSignImpl.getUnifiedSign("<instance: java.util.HashMap>", "<instance: java.util.HashMap>", "23781390", null, true)  
 6726 ms | InnerSignImpl.convertInnerBaseStrMap("<instance: java.util.Map, $className: java.util.HashMap>", "23781390", true)  
 6726 ms | <= "<instance: java.util.Map, $className: java.util.HashMap>"  
 6727 ms | InnerSignImpl.getMiddleTierEnv()  
 6727 ms | <= 0  
 6737 ms <= "<instance: java.util.HashMap>"

点击发送请求时，调用了InnerSignImpl.getUnifiedSign函数。但是输入参数和数据参数均为HashMap类型，结果中未显示具体内容。从结果输出中猜测frida-trace是通过对需要hook的函数在\_\_handlers\_\_下生成js文件，并调用js文件进行hook操作的，因此修改“handlers\mtopsdk.security.InnerSignImpl\getUnifiedSign.js”，使其能正确输出HashMap类型。

// \_\_handlers\_\_\mtopsdk.security.InnerSignImpl\getUnifiedSign.js  
  
{onEnter(log, args, state) {  
 // 增加了HashMap2Str函数，将HashMap类型转换为字符串  
 function HashMap2Str(params\_hm) {  
 var HashMap=Java.use('java.util.HashMap');  
 var args\_map=Java.cast(params\_hm,HashMap);  
 return args\_map.toString();  
 };  
 // 当调用函数时，输出函数参数  
 log(`InnerSignImpl.getUnifiedSign(${HashMap2Str(args[0])},${HashMap2Str(args[1])},${args[2]},${args[3]})`);  
 }, onLeave(log, retval, state) {  
 function HashMap2Str(params\_hm) {  
 var HashMap=Java.use('java.util.HashMap');  
 var args\_map=Java.cast(params\_hm,HashMap);  
 return args\_map.toString(); };  
 if (retval !== undefined) {  
 // 当函数运行结束时，输出函数结果  
 log(`<= ${HashMap2Str(retval)}`);  
 } }}

再次运行frida-trace，输出的结果已经可以看到具体内容了：

(py3) PS E:\TEMP\damai> frida-trace -U -j "\*InnerSignImpl\*!\*" 大麦  
 ......  
Started tracing 27 functions. Press Ctrl+C to stop.  
 /\* TID 0x15ab \*/  
 2653 ms InnerSignImpl.getUnifiedSign({data={"itemId":"719193771661","performId":"211232892","skuParamListJson":"[{\"count\":1,\"price\":36000,\"priceId\":\"251592963\"}]","dmChannel":"\*@damai\_android\_\*","channel\_from":"damai\_market","appType":"1","osType":"2","calculateTag":"0\_0\_0\_0","source":"10101","version":"6000168"}, deviceId=null, sid=13abe677c5076a4fa3382afc38a96a04, uid=2215803849550, x-features=27, appKey=23781390, api=mtop.damai.item.calcticketprice, utdid=ZF3KUN8khtQDAIlImefp4RYz, ttid=10005890@damai\_android\_8.5.4, t=1684828096, v=2.0},{pageId=, pageName=},23781390,null)  
 2654 ms | InnerSignImpl.convertInnerBaseStrMap("<instance: java.util.Map, $className: java.util.HashMap>", "23781390", true)  
 2655 ms | <= "<instance: java.util.Map, $className: java.util.HashMap>"  
 2655 ms | InnerSignImpl.getMiddleTierEnv()  
 2655 ms | <= 0  
 2662 ms <= {x-sgext=JA2qmBOxRVDxFRzca3r9BZibqJqvn7uerZOriayYu4mpnKCeoJiunKGZu5qqyfmaqJqhmvqYr5n8zPyJqImpmbvLrImomqidu5m7m7uYu5u7mLuYu5u7m7ubqYmtiaiJqImoiaiJqImoiaiJu8+7iaCf/cypnruaqJqomruau5j8y7uau4mgiaiJqInf6fDIu5o=, x-umt=+D0B/05LPEvOgwKIQ1x+SeV5wNE6NzOo, x-mini-wua=atASnVJw3vGX1Tw3Y/zDaVZkDUbLxOxtlUmgDOnIjMTBcMPMqQJLpnxoOWEL53Fq/OPcQZiMpDXWNvDz8UQkI5mtkZvIcDN1oxZnuH0M22LHKar4rnO/xm4LtAiniKgYtfgMGK3stXuCmvtE4raIhROimslSk7hCkxaL/DYuLzBLYwXmNyr9UZi1g, x-sign=azG34N002xAAK0H9KwNr3txWFMxzW0H7ROfkLQK+Db7ueJHktR/yP/0TcdPFzoYf36zd9lJYMsHCmYX3EcoFnJPMk2pxu0H7QbtB+0}

可以看到返回结果中包含了 x-sgext,x-umt,x-mini-wua,x-sign 等加密参数。至此，前面的一大堆分析也算有了小的收获。但对比流量分析结果中的发送参数，还是缺失了很多参数。下面我们继续跟踪，找出剩下的参数。

**跟踪 mtopsdk**

调研发现淘系的apk都包含mtopsdk，猜想会不会有公开的官方文档描述mtopsdk的使用方法，因此我们就找到了 【阿里云mtopsdk Android接入文档】 。其中介绍了请求构建的流程如下，重点关注了请求构建和发送的部分：

// 3. 请求构建  
// 3.1生成MtopRequest实例  
MtopRequest request = new MtopRequest();  
// 3.2 生成MtopBuilder实例  
MtopBuilder builder = instance.build(MtopRequest request, String ttid);  
// 4. 请求发送  
// 4.2 异步调用  
ApiID apiId = builder.addListener(new MyListener).asyncRequest();

因此不妨大胆一些，直接跟踪所有对mtopsdk中函数的调用。

(py3) PS E:\TEMP\damai> frida-trace -U -j "\*mtopsdk\*!\*" 大麦

输出的结果大概有2000行，直接看太费劲，复制到文本编辑器里做进一步分析。

按照阿里的官方文档介绍的流程，对应可以找到在输出的trace中找到一些关键的日志。

# MtopRequest初始化  
 3249 ms MtopRequest.$init()  
 3249 ms MtopRequest.setApiName("mtop.trade.order.build")  
 3249 ms MtopRequest.setVersion("4.0")  
 3249 ms MtopRequest.setNeedSession(true)  
 3249 ms MtopRequest.setNeedEcode(true)  
 3249 ms MtopRequest.setData("{\"buyNow\":\"true\",\"buyParam\":\"7191937661\_1\_51826442779\",\"exParams\":\"{\\\"atomSplit\\\":\\\"1\\\",\\\"channel\\\":\\\"damai\_app\\\",\\\"coVersion\\\":\\\"2.0\\\",\\\"coupon\\\":\\\"true\\\",\\\"seatInfo\\\":\\\"\\\",\\\"umpChannel\\\":\\\"10001\\\",\\\"websiteLanguage\\\":\\\"zh\_CN\_#Hans\\\"}\"}")  
  
# MtopBuilder初始化  
 3251 ms MtopBuilder.$init("<instance: mtopsdk.mtop.intf.Mtop>", "<instance: mtopsdk.mtop.domain.MtopRequest>", null)  
  
# MtopBuilder发送异步请求  
3268 ms MtopBuilder.asyncRequest()  
  
# 参数构建  
3301 ms | | | InnerProtocolParamBuilderImpl.buildParams("<instance: mtopsdk.framework.domain.MtopContext>")  
3391 ms | | | <= "<instance: java.util.Map, $className: java.util.HashMap>",{wua=CofS\_+7HCuvRCdz1EN8ICI6A4ZBCJwgY1hi+Bsivjcijs8GggmUxLQQUVTEQ5mYYtPuV7R2QNG5JEONIJRfmzjxFXMrs9AHdepIuqoJJJAyewWALprRnjIAu75t47Tm/RU9xRi7IEo9w0P2aCquLzf7uhiO8JEDSRK/ZdVhURBbof7reFtzEBoYYeIPgnwz7CL3kRlbyqyJcYKxO7ZmmVq1PtMXF2HGJqRSDjdv9l4mySJljIQzBmpX393L6eO1ZQVG1fpp6RaCRcFF+UgfjJXaeMFziHzfQF7KfUQZIeAJV/4GyVEE2f55RwPluOTuQubXQnq+qu41a0V5oyEOFXMoQRYFZzLOv3CjwkiIXsqJFeIHc=, x-sgext=JA0VLKcO8e9Fqqhj38VJuiwkHCUbIA8jGCwUNh0mDzYdIxQhFCcVJxskDyUedk0lHCUVJU4nGyZIc0g2HDYdJg90GDYcJRwiDyYPJA8kDyQPJA8kDyQPJA8nDyQPJQ8lDyUPJQ8lDyUPJQ82STYPLRlwSiUcNhwlHCUcNhw2HnFNNhw2Dy0PJQ8lD1JvfU42HA==, nq=WIFI, data={"buyNow":"true","buyParam":"719193771661\_1\_5182956442779","exParams":"{\"atomSplit\":\"1\",\"channel\":\"damai\_app\",\"coVersion\":\"2.0\",\"coupon\":\"true\",\"seatInfo\":\"\",\"umpChannel\":\"10001\",\"websiteLanguage\":\"zh\_CN\_#Hans\"}"}, pv=6.3, sign=azG34N002xAAKiYA2sv237H04abW2iYKIxaD3GVPak+JifYV0u6VzpriFiKiP+HuuF26BzWpVTClaOIGdjtibfQ99JomGiYKJhomCi, deviceId=null, sid=13abe677c5076a4fa3382afc38a96a04, uid=2215803849550, x-features=27, x-app-conf-v=0, x-mini-wua=a3gSvx5K5/NRy/W8+fDouCSQ6VSmMK3awHwo5X+IayY7JL5SwHtiL0soynSAvCobk01qRQ2fQcTvZWakhmhA9xlNOKdwvxdA5nZ4Tno2asO5e7EvSMj6yqVYAXZZUBjZPUOBw3vpH8L2GUq9Gi6MTszU57a58+hJE2BCGTVsxhRonDw1Nnxp74Ffm, appKey=23781390, api=mtop.trade.order.build, umt=+D0B/05LPEvOgwKIQ1x+SeV5wNE6NzOo, f-refer=mtop, utdid=ZF3KUN8khtQDAIlImefp4RYz, netType=WIFI, x-app-ver=8.5.4, x-c-traceid=ZF3KUN8khtQDAIlImefp4RYz1684829318230001316498, ttid=10005890@damai\_android\_8.5.4, t=1684829318, v=4.0, user-agent=MTOPSDK/3.1.1.7 (Android;9;samsung;SM-S908E)}

注意到InnerProtocolParamBuilderImpl.buildParams函数的输出结果完全覆盖了需要的各类加密参数，其输入类型是MtopContext。从jadx逆向的apk代码中可以找到MtopContext类，即包含Mtop生命周期的各个类的一个容器。

public class MtopContext {  
 public ApiID apiId;  
 public String baseUrl;  
 public MtopBuilder mtopBuilder;  
 public Mtop mtopInstance;  
 public MtopListener mtopListener;  
 public MtopRequest mtopRequest;  
 public MtopResponse mtopResponse;  
 public Request networkRequest;  
 public Response networkResponse;  
 public MtopNetworkProp property = new MtopNetworkProp();  
 public Map<String, String> protocolParams;  
 public Map<String, String> queryParams;  
 public ResponseSource responseSource;  
 public String seqNo;  
 @NonNull  
 public MtopStatistics stats;  
}

所以现在的问题变为如何能够构建出来MtopContext，然后调用buildParams函数生成各类加密参数。

**分析业务模块与mtopsdk的交互过程**

在写本文复盘分析过程的时候，发现仅依赖mtopsdk的调用过程其实已经可以得到MtopContext的全部生成逻辑了。但所谓当局者迷，在当时分析的时候还是一头雾水。因此在此也介绍一下这里的思考逻辑。

当时看着mtopsdk的调用过程，感觉很复杂。但是猜想从用户点击操作->业务代码->mtopsdk的数据流，以及模块间高内聚低耦合的原则，所以猜想模块间的调用不会很复杂，所以就想分析业务代码与mtopsdk的调用逻辑。所以就想跟踪主要业务代码的trace。所以继续跟踪trace，运行frida-trace -U -j "\*cn.damai\*!\*" 大麦 ，以分析cn.damai包的调用过程，在其中发现了 NcovSkuFragment.buyNow() 函数，看起来是和购买紧密相关的函数。又找到DMBaseMtopRequest类。

但是在这里有点卡住了，因为只找到了构建MtopRequest，并未在cn.damai的trace日志中并未发现其他对mtop的调用。

然后又尝试搜索和api(order.build)相关的代码，然而并没有多大用处。

然后，又读了大量的源代码，终于定位到了 com.taobao.tao.remotebusiness.MtopBussiness这个关键类。

本以为com.taobao开头的代码不是那么重要，所以最开始把这个类完全忽略了。但通过对源码的阅读，发现这个类是motpsdk中MtopBuilder类的子类，主要负责管理业务代码和Mtopsdk的交互。

因此继续通过trace跟踪MtopBussiness类。运行frida-trace -U -j "\*!\*buyNow\*" -j "com.taobao.tao.remotebusiness.MtopBusiness!\*" -j "\*MtopContext!\*" -j "\*mtopsdk.mtop.intf.MtopBuilder!\*" 大麦

现在业务代码和mtopsdk的交互就很清晰了

**六、hook得到接口参数**

通过以上对trace的分析，已经知道了具体执行的操作，因此可以使用frida编写js代码，直接调用APK中的类，实现功能调用。

先展示一个简单的示例，用于构建一个自定义的MtopRequest 类:

// new\_request.js  
Java.perform(function () {  
 const MtopRequest = Java.use("mtopsdk.mtop.domain.MtopRequest");  
 let myMtopRequest = MtopRequest.$new();  
 myMtopRequest.setApiName("mtop.trade.order.build");  
 //item\_id + count + ski\_id 716435462268\_1\_5005943905715  
 myMtopRequest.setData("{\"buyNow\":\"true\",\"buyParam\":\"716435462268\_1\_5005943905715\",\"exParams\":\"{\\\"atomSplit\\\":\\\"1\\\",\\\"channel\\\":\\\"damai\_app\\\",\\\"coVersion\\\":\\\"2.0\\\",\\\"coupon\\\":\\\"true\\\",\\\"seatInfo\\\":\\\"\\\",\\\"umpChannel\\\":\\\"10001\\\",\\\"websiteLanguage\\\":\\\"zh\_CN\_#Hans\\\"}\"}")  
 myMtopRequest.setNeedEcode(true);  
 myMtopRequest.setNeedSession(true);  
 myMtopRequest.setVersion("4.0");  
 console.log(`${myMtopRequest}`)  
});

再使用运行命令 frida -U -l .\reverse\new\_request.js 大麦，以在该APP的Apk中执行js hook代码。运行之后即可输出自己构建的MtopRequest实例。（frida真的很奇妙！）

有了上面的结果，下面继续完善这个示例，添加MtopBussiness的构建过程和输出过程

//引入Java中的类  
 const MtopBusiness = Java.use("com.taobao.tao.remotebusiness.MtopBusiness");  
 const MtopBuilder = Java.use("mtopsdk.mtop.intf.MtopBuilder");  
 // let RemoteBusiness = Java.use("com.taobao.tao.remotebusiness.RemoteBusiness");  
 const MethodEnum = Java.use("mtopsdk.mtop.domain.MethodEnum");  
 const MtopListenerProxyFactory = Java.use("com.taobao.tao.remotebusiness.listener.MtopListenerProxyFactory");  
 const System = Java.use('java.lang.System');  
 const ApiID = Java.use("mtopsdk.mtop.common.ApiID");  
 const MtopStatistics = Java.use("mtopsdk.mtop.util.MtopStatistics");  
 const InnerProtocolParamBuilderImpl = Java.use('mtopsdk.mtop.protocol.builder.impl.InnerProtocolParamBuilderImpl');  
  
 // create MtopBusiness  
 let myMtopBusiness = MtopBusiness.build(myMtopRequest);  
 myMtopBusiness.useWua();  
 myMtopBusiness.reqMethod(MethodEnum.POST.value);  
 myMtopBusiness.setCustomDomain("mtop.damai.cn");  
 myMtopBusiness.setBizId(24);  
 myMtopBusiness.setErrorNotifyAfterCache(true);  
 myMtopBusiness.reqStartTime = System.currentTimeMillis();  
 myMtopBusiness.isCancelled = false;  
 myMtopBusiness.isCached = false;  
 myMtopBusiness.clazz = null;  
 myMtopBusiness.requestType = 0;  
 myMtopBusiness.requestContext = null;  
 myMtopBusiness.mtopCommitStatData(false);  
 myMtopBusiness.sendStartTime = System.currentTimeMillis();  
  
 let createListenerProxy = myMtopBusiness.$super.createListenerProxy(myMtopBusiness.$super.listener.value);  
 let createMtopContext = myMtopBusiness.createMtopContext(createListenerProxy);  
 let myMtopStatistics = MtopStatistics.$new(null, null); //创建一个空的统计类  
 createMtopContext.stats.value = myMtopStatistics;  
 myMtopBusiness.$super.mtopContext.value = createMtopContext;  
 createMtopContext.apiId.value = ApiID.$new(null, createMtopContext);  
  
 let myMtopContext = createMtopContext;  
 myMtopContext.mtopRequest.value = myMtopRequest;  
 let myInnerProtocolParamBuilderImpl = InnerProtocolParamBuilderImpl.$new();  
 let res = myInnerProtocolParamBuilderImpl.buildParams(myMtopContext);  
 console.log(`myInnerProtocolParamBuilderImpl.buildParams => ${HashMap2Str(res)}`)

再次执行frida -U -l .\reverse\new\_request.js 大麦，此时已能根据笔者任意构建的请求data输出其他加密参数

对于order.create的原理类似，此处不再赘述。

**补充说明**

通过frida调用Apk中的Java类有时候会出现找不到类的情况，原因可能是classloader没有正确加载。可以在js代码前的最前面加上下面的代码，指定正确的classloader，即可解决该问题

Java.perform(function () {  
 //get real classloader  
 //from http://www.lixiaopeng.top/article/63.html  
 var application = Java.use("android.app.Application");  
 var classloader;  
 application.attach.overload('android.content.Context')  
 .implementation = function (context) {  
 var result = this.attach(context); // run attach as it is  
 classloader = context.getClassLoader(); // get real classloader  
 Java.classFactory.loader = classloader;  
 return result;  
 }  
});

**七、通过rpc调用**

前文提到frida的一个特性是可以通过rpc调用提供python编程接口。

一个简单的示例：

import frida  
  
def on\_message(message, data):  
 if message["type"] == "send":  
 print("[\*] {0}".format(message["payload"]))  
 else:  
 print(message)  
  
# hook代码  
jscode = """  
rpc.exports = {  
 testrpc: function (a, b) { return a + b; },  
}; """  
  
def start\_hook():  
# 开始hook  
 process = frida.get\_usb\_device().attach("大麦")  
 script = process.create\_script(jscode)  
 script.on("message", on\_message)  
 script.load()  
 return script  
  
script = start\_hook()  
# 调用hook代码  
print(script.exports.testrpc(1, 2))  
  
# >>> 输出  
# 3

frida使用rpc的方法也很简单，仅需使用rpc.exports，将对应的函数暴露出来，就能被python调用。

完整的代码就是将上一章的代码封装为函数，并通过rpc对外提供接口，就可以了。为避免侵权，本文不贴出完整利用代码。

代码封装完成后测了一下，平均一次调用的时间为0.024秒，完全可以达到抢票的要求。

**八、踩坑花絮**

**关于wiresharkhelper**

txthinking放出了一个抓包辅助工具wiresharkhelper，看视频介绍很诱人很方便，但是实测是要收费的。本人穷，所以就没用他的方法。然而也是因为这个才开始尝试用frida工具得到https的密钥，发现了frida这个神器。

**关于Cookie**

发送的请求头里是包含cookie的，但是本文没有介绍。其实本来是再继续找cookie的，但是发现把InnerProtocolParamBuilderImpl.buildParams函数的参数填进去之后，就已经能正常获取服务器的返回值了，所以就没继续搞cookie

**关于MtopStatistics**

MtopStatistics是mtopsdk里比较重要的一个类，用来跟踪用户的操作记录状态，可能有助于判断用户是否是机器人。但尝试自己构建MtopStatistics失败，所以直接生成了一个空的MtopStatistics类，好在也没对服务器的正常返回造成影响。

**如何获取票价信息**

这里笔者是直接用的某麦网Web端PC版，网页中有一段json，包含静态的描述信息和动态的场次、余票信息。

**如何脱离模拟器运行**

目前是需要模拟器一直运行着的，而且仅能用一个人的账户。这对于个人使用是完全够用的。如何能脱离模拟器，而且增加并发用户数量还需要继续研究。目前时间不允许，暂时不再继续此问题的研究。

**九、总结**

本文完整的记录了对于Apk与服务器交互API的解析过程，包括环境搭建、抓包、trace分析、hook、rpc调用。本文对于淘系Apk的分析可以提供较多参考。本文算是本人第一次深入且成功的用动态+静态分析结合的方式，借助神器frida+jadx，成功破解Apk，因此本文的记录也较为细致的记录了思考过程。

本文也有一些不足之处，如无法脱离模拟器运行、仅能单用户等。