

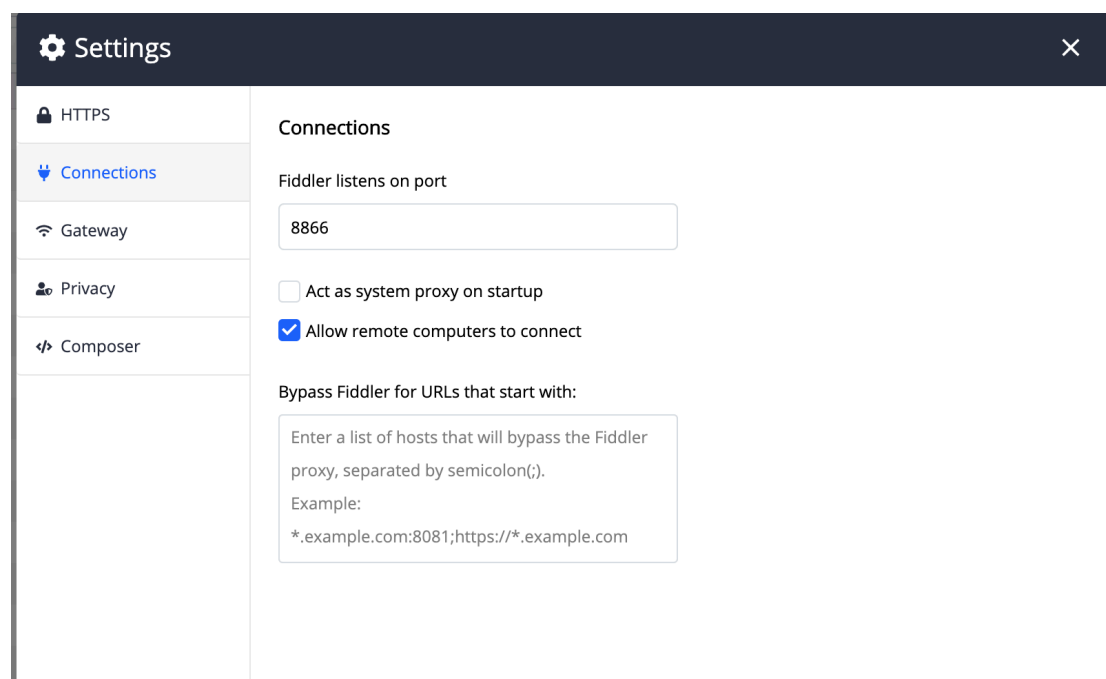
IOS 版京东 App 抢茅台从原理到实现

从 2 月份开始，原始通过 python 脚本调用 PC 版本网页实现抢茅台的功能已经不可用，根据网页显示数据来看，从 2 月份开始，只能通过京东 APP 抢购。

开始分析

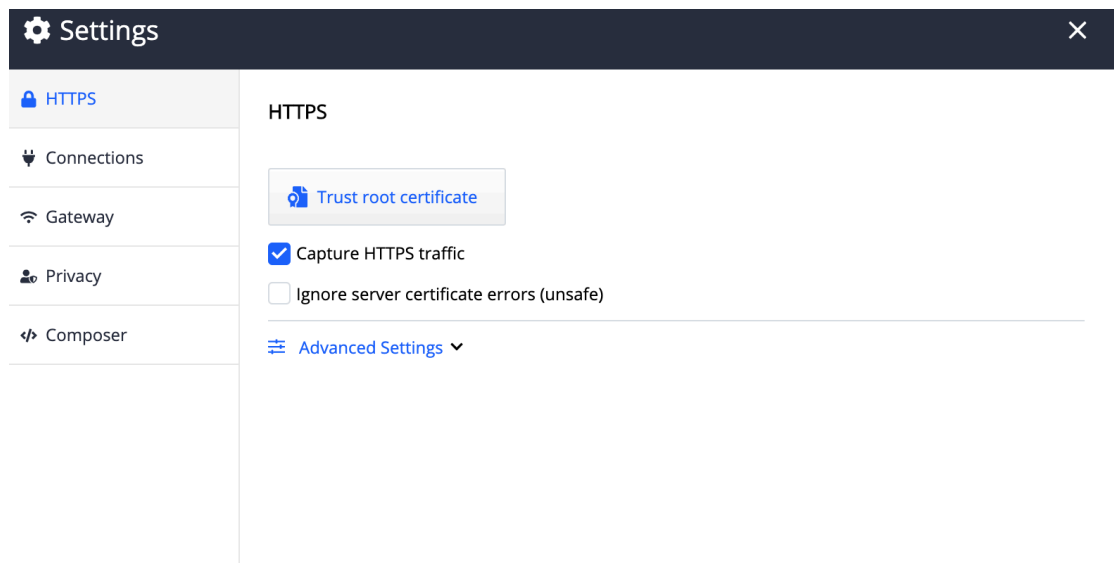
第一步 抓包

使用 Fiddler 抓包，在 PC 端安装 Fiddler 软件，配置好选项，主要需要关注的位置如下：





端口处设置端口号码，打开允许远程电脑访问（因为我们等会需要在手机上来访问这个代理）

此外需要打开 HTTPS 解密功能。











回到手机上，使用手机自带的 Safari 浏览器访问 PC 地址加上设置的端口号，来下载 TLS 根证书，我这边地址是 <http://192.168.4.110:8866/>，被证书下载回来之后，到手机的设置里面可以已经下载描述文件，点进去可以安装，安装完成后，还需要通用->关于本机->证书信任设置里面把证书信任了，最后，到无线局域网下面找到手机和 PC 所在的网络，手动设置好代理地址和端口号。

到此环境已经就绪，打开京东 APP，找到茅台页面，开始执行预约操作，到时间点后，手动抢一波（当然是没抢到的啦，要是这么容易抢到就不用这个分析了），一会儿就抢完之后就可以在 Fiddler 里面看到请求的日志信息了，通过观察日志，预约相关的请求有如下两个：

11:58:27.530		97	api.m.jd.com	https://api.m.jd.com/client.action?functionId=isAppoint
11:58:28.170		98	api.m.jd.com	https://api.m.jd.com/client.action?functionId=appoint

第一个判断是否有预约，第二个执行预约。

秒杀的流程大概如下：

12:00:03.227		109	api.m.jd.com	https://api.m.jd.com/client.action?functionId=lv
12:00:03.918		110	api.m.jd.com	https://api.m.jd.com/client.action?functionId=isAppoint
12:00:04.547		113	api.m.jd.com	https://api.m.jd.com/client.action?functionId=genToken
12:00:04.829		115	un.m.jd.com	https://un.m.jd.com/cgi-bin/app/appjmp?tokenKey=AAEAMGa_jK1_eHII5I6C
12:00:05.172		117	divide.jd.com	https://divide.jd.com/user_routing?skuld=100012043978&mid=Q-WH7YZf5:
12:00:05.514		119	marathon.jd.com	https://marathon.jd.com/m/captcha.html?sid=212395f239829b5f0de354e2
12:00:05.583		120	marathon.jd.com	https://marathon.jd.com/seckillM/seckill.action?skuld=100012043978&nur
12:00:06.007		121	marathon.jd.com	https://marathon.jd.com/seckillnew/orderService/init.action

如果之前分析过 PC 版的逻辑，就会发现很熟悉的部分 <https://marathon.jd.com/seckillnew/orderService/init.action>

总结一下抢购的流程

1. 访问 <https://api.m.jd.com/client.action?functionId=lv> 获取路由地址
2. 访问 <https://api.m.jd.com/client.action?functionId=genToken> 获得 token
3. 访问 <https://un.m.jd.com/cgi-bin/app/appjmp?tokenKey=&lbs=&to=> , 这个接口有 3 次 302 跳转, 第一次跳转到 https://divide.jd.com/user_routing?(参数省略, 下同), 第二次跳转到 <https://marathon.jd.com/m/captcha.html>, 第三次跳转到 <https://marathon.jd.com/seckillM/seckill.action>, 每次跳转前都会设置一些 Cookie (http 头 Set-Cookie)。
4. 第三步执行完成, 会加载一段 js 脚本, 然后客户端就看到提交订单页面
5. 点击提交订单之后,
 - a) 会有初始化订单 :
<https://marathon.jd.com/seckillnew/orderService/init.action>
 - b) 计算价格 :
<https://marathon.jd.com/seckillnew/orderService/calcuOrderPrice.action>
 - c) 京东豆查询 :
<https://marathon.jd.com/seckillnew/jBeanService/getJBeanInfo.action>这几个请求
6. 最后调用下单接口 :
<https://marathon.jd.com/seckillnew/orderService/submitOrder.action>

第二步 抓包分析

拿到抓包数据之后, 你可能就会开始尝试模拟发包, 可是, 打开京东 APP 的请求数据, 准备构造请求, 立马傻眼了, 以 genToken 接口为例, 他上传的请求参数有很多个, 大部分都通俗易懂, 且是明文, 部分如下 :

body	{"to":"https://divide.jd.com/user_routing?skuld=100012043978","action":"to"}
build	167541
client	apple
clientVersion	9.4.0
d_brand	apple
d_model	iPhone9,2

拉到最后, 发现 3 个参数, 一个签名信息, 一个时间戳, 一个签名版本 (先这么

认为，最后分析 APP 会发现不是所谓的版本号)

sign	79ad88422c8725724edf11411c81f583
st	1612756881782
sv	121

多抓几次包，就会发现，即便其它字段相同，由于 st 和 sv 不同也会导致 sign 不同，很显然这两个值应该都参与了 sign 的计算。通过抓包过程的分析，你就会发现，其实目前的流程来看，所有的难点现在都集中到 sign 的计算上，无法搞定 sign，所有的功能无从谈起。

第三步 查找 sign

首先在网上搜索看看，有没有前辈解决了这个问题，首先找到这个文章，<https://blog.csdn.net/hamy168/article/details/106316807/>，这篇文章总结了 android 版本参与签名的参数信息，本来我手上的 iPhone，但是目前并没有 root，所以还是从官网下载了 android 版本的 app 进行了简单分析，发现流程和上述文章说的一样，在 android 的系统下面，则只需要注入到京东 app，模拟手机发起调用就可以拿到 sign 参数了，但是对于我的 IOS 设备来说，根本就行不通，继续在网上搜索资料，几个小时过去之后，发现了最大的突破口，突破口是这位博主：<http://91fans.com.cn/post/jdsignone/>的网站，这位博主已经找到了计算 sign 的 so，并且通过 AndroidNativeEmu 模拟执行 so 的方法来实现 PC 上计算 sign，大神并没有给出代码，通过研究了一下他博客中相关的文章，发现目前能够模拟执行 so 的项目主要有两个，AndroidNativeEmu 和 unidbg，AndroidNativeEmu 主要使用 python 实现，unidbg 使用 java 实现，考虑到自己并不是 python 的忠实粉丝，所以开始学习 unidbg 的源码。而且从他的文章了解到 android 版本计算 sign 的逻辑位于 libjdbitmapkit.so，(嗯，这个文件够有迷惑性，bitmap 和 sign 有啥关系？!) 使用 ghidra (一个类似 IDA 工具，开源) 打开 libjdbitmapkit.so，找到相关计算签名的方法，完全逆向出来难度太大，而且总算明白的 sv 是啥，sv 是一个随机数，但是随机后模 3，第一位不可能是 0，所以随机后就那么几个值；100,101,102,110,111,112,120,121,122

Unidbg 跑 so 文件

项目位于 <https://github.com/zhkl0228/unidbg>, 从更新时间来看, 这个项目还是很活跃的, 开始下载回来跑 android 版本的 libjdbitmapkit.so, 根据查看 test 下面的项目, 发现自己跑 so 只需要继承一个 AbstractJn 就可以了, 框架已经实现了模拟包名, 签名信息等那些基础功能, 我们需要做的就是将京东 APP 的相关信息设置进去, 然后开始执行。中途出现多个错误, 大部分都是 so 里面回调 java 层的方法失败, 所以这里需要了解一下 AbstractJn 的函数, public DvmObject<?> newObjectV(BaseVM vm, DvmClass dvmClass, String signature, VaList vaList) 这个函数在 so 创建 java 对象的时候被调用, public DvmObject<?> callObjectMethodV(BaseVM vm, DvmObject<?> dvmObject, String signature, VaList vaList), 这个方法在在 java 对象上调用方法的时候被调用, public DvmObject<?> getStaticObjectField(BaseVM vm, DvmClass dvmClass, String signature)这个方法在调用静态成员的的时候被调用, 执行 so 的时候, 需要的大部分错误为 XXX Object 没有找到, XXX 方法没有找到, 所以只需要自己重载上面这个接口, 实现相应的对象创建和功能实现就可以了, 对于 libjdbitmapkit 来说, 我们重载需要实现如下几个接口

```
@Override
public DvmObject<?> newObjectV(BaseVM vm, DvmClass dvmClass, String signature, VaList vaList){
    if (logging){
        System.out.println(signature);
    }
    switch (signature){
        case "java/lang/Integer-><init>(I)V":
            int value = vaList.getInt( offset: 0);
            return vm.resolveClass( className: "java/lang/Integer").newObject(new Integer(value));
        case "java/lang/StringBuffer-><init>()V":
            return vm.resolveClass( className: "java/lang/StringBuffer").newObject(new StringBuffer());
    }
    return super.newObjectV(vm, dvmClass, signature, vaList);
}
```

```

@Override
public DvmObject<?> callObjectMethodV(BaseVM vm, DvmObject<?> dvmObject, String signature, VaList vaList)
{
    if (logging){
        System.out.println(signature);
    }
    switch (signature) {
        case "java/lang/StringBuffer->append(Ljava/lang/String;)Ljava/lang/StringBuffer;": {
            StringBuffer object = (StringBuffer) dvmObject.getValue();
            String sub = vaList.getObject( offset: 0).getValue().toString();
            return vm.resolveClass( className: "java/lang/StringBuffer").newObject(object.append(sub));
        }
        case "java/lang/Integer->toString()Ljava/lang/String;": {
            Integer intVal = (Integer) dvmObject.getValue();
            return vm.resolveClass( className: "java/lang/String").newObject(intVal.toString());
        }
        case "java/lang/StringBuffer->toString()Ljava/lang/String;": {
            StringBuffer object = (StringBuffer) dvmObject.getValue();
            return vm.resolveClass( className: "java/lang/String").newObject(object.toString());
        }
    }
    return super.callObjectMethodV(vm,dvmObject,signature,vaList);
}

```

```

@Override
public DvmObject<?> getStaticObjectField(BaseVM vm, DvmClass dvmClass, String signature){
    if (signature.equals("com/jingdong/common/utils/BitmapkitUtils->a:Landroid/app/Application;")){
        return packageManger;
    }
    return super.getStaticObjectField(vm,dvmClass,signature);
}

```

最后调用签名接口

```

String getSign(String functionID, String body, String uuid, String client, String clientVersion) {
    DvmObject context = vm.resolveClass( className: "android/content/Context").newObject( value: "and
    String method = "getSignFromJni(Landroid/content/Context;Ljava/lang/String;Ljava/lang/String;I
    DvmObject str1 = new StringObject(vm,functionID);
    DvmObject str2 = new StringObject(vm,body);
    DvmObject str3 = new StringObject(vm,uuid);
    DvmObject str4 = new StringObject(vm,client);
    DvmObject str5 = new StringObject(vm,clientVersion);
    DvmObject ret = BitmapkitUtils.callStaticJniMethodObject(emulator,
        method,context, str1, str2, str3, str4, str5); // 执行Jni方法
    return ret.getValue().toString();
}

```

不出意外，然后就可以得到正确的 sign 了。搞定 sign 的计算，用 IOS 平台的数据替换 android 的数据，开始模拟发包，调用 lv 接口，立马就收到签名不正确的返回值，这下又傻眼了，到底是跑出来的 sign 不正确还是 IOS 和 android 的签名不一样？为了验证 android 的 sign 是否正确，找到网上的测试数据，我们需要 hook 相应的函数，返回时间戳。

```

void setupHook(){
    IxHook xHook = XHookImpl.getInstance(emulator); // 加载xHook, 支持Import hook, 支持
    xHook.register( pathname_regex_str: "libjdbitmapkit.so", symbol: "gettimeofday", n
        Pointer pointer1 = null;
        @Override
        public HookStatus onCall(Emulator<?> emulator, HookContext context, long c
            pointer1 = context.getPointerArg( index: 0);
            // String str = pointer.getString(0);
            System.out.println("gettimeofday");
            // context.push(str);
            return HookStatus.RET(emulator, originFunction);
        }
        @Override
        public void postCall(Emulator<?> emulator, HookContext context) {
            // Pointer pointer = context.getPointerArg(0);
            if(pointer1 != null){
                // 这里把 时间写死
                byte[] buf = {(byte)0x91,(byte)0x50,(byte)0xc4,(byte)0x5f,(byte)0x
                pointer1.write( offset: 0,buf, index: 0, length: 8);
            }
            // ByteBuffer tv_sec = pointer1.getByteBuffer(0,8);
            System.out.println("gettimeofday 0k"); // + context.pop() ); // + '
        }
    }, enablePostCall: true);
    xHook.refresh();
}

```

然后跑一下，发现和网上的验证数据一致，那接下来问题就来了，说明 IOS 和 android 有差异，通过检查 IOS 的数据，发现 uuid 是个 base64 编码的 string，Android 却是类似这样的结构 99001184062989-f460e22c02fa，然后立马测试 uuid 传 hex.decode(base64.decode(uuid)) 结果不对，继续看 ios 的参数，有个 adid 和 openudid 和 android 的 uuid 很像，传进去试试，也不对。在使用 unidbg 的时候，发现它对 IOS 的二进制有实验性的支持，到目前还看，已经没有其它路可以走了，必须要对京东 APP IOS 版本进行逆向分析了，没办法，自己这边用的是 iPhone,总不能为了抢茅台去搞个 android 机器吧？

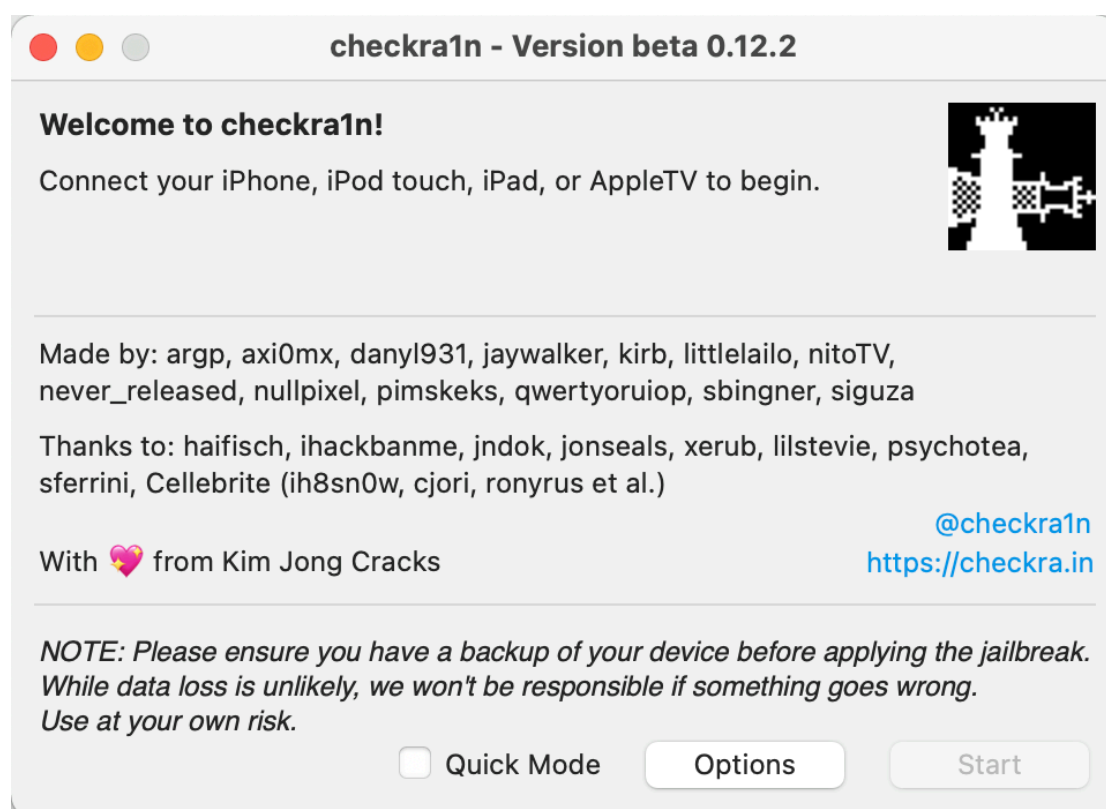
寻找砸壳的 IPA 包

根据前述的分析，可以确定 IOS 版本和 android 版本 sign 签名是有差异的，所以流程往下走，就不得不进行 IOS 版本的 APP 逆向分析了，而 IOS 版本逆向分析的前提就是获取砸壳后的 IPA 包，普通的 APP 可以在已经 Root 的机器上使用

frida 砸壳脚本砸壳，但是由于自己两台 iPhone，一台 7P，一台 12 pro max 均在 IOS 14.3 版本，暂时没有一键 root 的方法，自己砸壳的方案先不选择。考虑到之前有很多第三方助手例如爱思助手，PP 助手本身有很多砸壳后的 IPA 包可以下载，立马去官网看看，发现 PP 助手基本已经 OVER 了，而爱思助手现在下载的京东 APP 只有共享正版，也是带壳的，百度一圈没有找到砸壳后的 APP，所以看来还是只能自己越狱自己的手机，然后自己砸壳了。

IOS14.3 越狱

爱思助手自带的一键 root 已经不支持 IOS14.3 了，而且现在我的 7P 系统降级也是没有通道的，所以只能寻找手动越狱工具。搜索一番之后找到，<https://checkra1n/>，这个工具使用 A 处理器漏洞实现临时越狱（重启后失效），可以查看自己的设备是否支持越狱，我的 7P 恰好可以，然后下载回来，连上手机开始越狱：



软件界面有一步步指导如何操作的提示，按照提示即可越狱成功，越狱成功后增加一个 checkra1n（图标和 PC 版本一样）的 app，从这个 app 进去就可以选择

安装 cydia 了，安装完 cydia，然后安装 frida app，具体可以参看官方文档：

<https://frida.re/docs/ios/#with-jailbreak>

注意同时用 cydia 安装好 OpenSSH，这样我们可以方便从 PC 上访问手机，接下来下载砸壳脚本 <https://github.com/AloneMonkey/frida-ios-dump>

然后打开 dump.py,设置手机 ssh 信息

```
User = 'root'
```

```
Password = 'alpine'
```

```
Host = '192.168.4.103'
```

```
Port = 22
```

```
KeyFileName = None
```

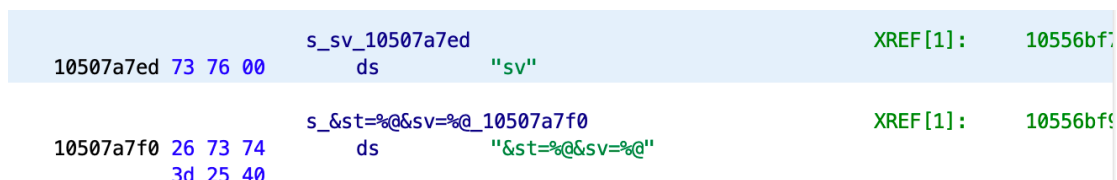
USB 连接上手机，打开京东 APP，PC 上执行 ./dump.py 京东 等一会儿就在当前目录得到了砸壳后的 IPA 包了：



砸壳后，使用 ghidra 打开 JD4IPhone，然后开始漫长的自动化分析（大概 4 小时），这期间继续学习 unidbg ios 部分的代码。

砸壳后 JD4IPhone 逆向分析

在长达 4 小时的分析之后，ghidra 自动化分析完成，打开搜索框，搜索字符串 sign,发现有很多字符串，改成 sign=, 未搜索到，改成 st=,找到如下位置：



转到引用，发现被函数 JDSignService::getSignWithDic:keys:引用，看起来这个函数就是计算签名的位置了，继续向上找引用这个函数的位置：

```

10058bc78 01 1f 18 4e    mov     v1.D[0x1],x24
10058bc7c e0 87 02 ad    stp     q0,q1,[sp, #s_functionId]
10058bc80 fa 3b 00 f9    str     x26=>cf_clientVersion,[sp, #s_clientVersion]    "clientVersion",00
10058bc84 08 1f 03 b0    adrp    x8,PTR_s_doSimpleHttpPost:body:_10696c000        = 104d00cd0
10058bc88 00 69 43 f9    ldr     dic=>_OBJC_CLASS_$_NSArray,[x8, #0x6d0]=>_OB... = 1073e62c0
                                                    = ??
                                                    = 1049a587a
10058bc8c 88 18 03 90    adrp    x8,PTR_s_containerView_10689b000
10058bc90 01 b1 42 f9    ldr     param_2=>s_arrayWithObjects:count:_1049a6ac4,[... = "arrayWithObjects:count:"
                                                    = 1049a6ac4

10058bc94 e2 43 01 91    add     x2,sp,#0x50
10058bc98 a3 00 80 52    mov     w3,#0x5
10058bc9c c7 93 0e 95    bl      __stubs::_objc_msgSend                        undefined _objc_msgSend()
10058bca0 e3 03 00 aa    mov     x3,keys
10058bca4 08 1f 03 b0    adrp    x8,PTR_PTR_10696d000                            = 1069fb270
10058bca8 00 cd 46 f9    ldr     keys=>objc::class_t::JDSignService,[x8, #0xd98... = 106b0f990
                                                    =
                                                    = 1049d77c8
10058bcac e8 18 03 90    adrp    x8,PTR_s_extraCookie_1068a7000
10058bc90 01 f9 40 f9    ldr     param_2=>s_getSignWithDic:keys:_1049d7e3a,[x8,... = "getSignWithDic:keys:"
                                                    = 1049d7e3a

10058bcb4 e2 03 14 aa    mov     x2,x20
10058bcb8 c0 93 0e 95    bl      __stubs::_objc_msgSend                        undefined _objc_msgSend()
10058bcbc 02 00 00 14    b       LAB_10058bcc4

```

查看这个函数转过来的 C 语言视图：

```

dic = __stubs::_objc_msgSend
      (&_OBJC_CLASS_$_NSDictionary,"dictionaryWithObjectsAndKeys:",lVar2);
s_body = &cf_body;
s_functionId = &cf_functionId;
s_client = &cf_client;
s_openudid = &cf_openudid;
s_clientVersion = &cf_clientVersion;
keys = __stubs::_objc_msgSend
      (&_OBJC_CLASS_$_NSArray,"arrayWithObjects:count:",&s_functionId,5);
lVar2 = __stubs::_objc_msgSend
      (&objc::class_t::JDSignService,"getSignWithDic:keys:",dic,keys);
goto LAB_10058bcc4;

```

根据此处确定 JDSignService::getSignWithDic:keys 的两个参数，第一个参数是一个 NSDictionary，第二个参数是 NSArray，而且这个数据初始化的时候使用下面的字符串数组：

"functionId", "body", "openudid", "client", "clientVersion"

由此可以看出参加签名的字典 key。由于 HTTP 请求的字段值是字符串，所以猜测这个用于签名的字典值类型是 NSString。那它的返回值是什么呢？回到函数 JDSignService::getSignWithDic:keys，根据 C 语言视图的流程：

```

dateNow = __stubs::_objc_retainAutoreleasedReturnValue();
__stubs::_objc_msgSend(&objc::class_t::JDCAHashManager,"md5:",dateNow);
uVar5 = __stubs::_objc_retainAutoreleasedReturnValue();
__stubs::_objc_msgSend(mudic,"setValue:forKey:",uVar5,&cf_sign);
uVar6 = __stubs::_objc_msgSend(mudic,"copy");
__stubs::_objc_release(uVar5);
__stubs::_objc_release(dateNow);

```

可以看出，最后返回的也是一个字典，值类型是 NSString。接下来对真机进行 Hook 抓参数验证猜测，继续使用 frida 工具注入 js 进行 hook，打印出函数调用参数和返回值。

```

var className = "JDSignService";
var funcName = "+ getSignWithDic:keys:";
var hook = eval('ObjC.classes.' + className + '[' + funcName + ']');
Interceptor.attach(hook.implementation, {
    onLeave: function(retval) {

        console.log("[*] Class Name: " + className);
        console.log("[*] Method Name: " + funcName);
        console.log("\t[-] Type of return value: " + typeof retval);
        console.log("\t[-] Original Return Value: " + retval);
        var dict = ObjC.Object(retval);
        var enumerator = dict.keyEnumerator();
        var key;
        while ((key = enumerator.nextObject()) !== null) {
            var value = dict.objectForKey_(key);
            console.log(key.toString()+"---->"+value.toString());
        }
    },

    onEnter: function(args){

        var className = ObjC.Object(args[0]);
        var methodName = args[1];

        console.log("className: " + className.toString());
        console.log("methodName: " + methodName.readUtf8String());

        var dict = ObjC.Object(args[2]);
        var enumerator = dict.keyEnumerator();
        var key;
        while ((key = enumerator.nextObject()) !== null) {
            var value = dict.objectForKey_(key);
            console.log(key.toString()+"---->"+value.toString());
        }
        console.log("-----keyList-----")
        var array = new ObjC.Object(args[3]);
        var count = array.count().valueOf();
        for (var i = 0; i !== count; i++) {
            var element = array.objectAtIndex_(i);
            console.log(element.toString());
        }
    }
});

```

保持京东在前台，执行:

frida -U -l hook.js 京东

执行完成，随便操作下京东，可以看到监控到函数调用的信息

```
body---->{"pageNum":"1","lid":"19,1666,36267,36272"}
functionId---->followV3EnterMainPage
clientVersion---->9.4.0
client---->apple
openudid---[REDACTED]
-----keyList-----
functionId
body
openudid
client
clientVersion
[*] Class Name: JDSignService
[*] Method Name: + getSignWithDic:keys:
    [-] Type of return value: object
    [-] Original Return Value: 0x282aec500
st---->1612947209073
sv---->111
sign---->b3b9f901b9fe0bcfca41f24ab5b9a140
```

经过确认，我们静态分析的结论完全正确，而且参数类型和返回值也没有问题，返回的字典只有 3 个字段，正是模拟发包需要的 3 个字段。

unidbg 执行 JD4IPhone

静态分析已经找到了签名实现的函数，现在开始使用 unidbg 加载 JD4IPhone 之后自己调用 JDSignService::getSignWithDic:keys 来获得签名数据，考虑到从 unidbg 调用 Objec 函数比较麻烦，这里给 unidbg 原先自带的 object-dump 增加新功能，用于辅助生成 NSArray 和 NSDictionary 对象，于是给 unidbg-ios/src/main/native/ios/class-dump.h 增加一个辅助类：

```

@interface ObjcHelper : NSObject
+ (NSString*) NewString:(const char*)src;
+ (NSArray*) NewStringArray:(const char*)src :(const char*) separator;
+ (NSMutableDictionary*) SetDictionaryValue:(NSMutableDictionary*)dic :(const char*)key :(const char*)value;
+ (NSDate)NewDate:(const double )timeIntervalSince1970;
@end

```

然后实现相应的函数，这样方便我们生成字符串，字符串数组以及操作字符串类型的字典。这里简单提一下 unidbg 调用 objc 对象方法的标准流程，由于 unidbg 已经准备好了 objc 的运行时环境了，所以一旦二进制加载完毕之后，里面的所有 objc 的对象和类都可以使用，包括系统的类，例如 NSDictionary。使用的流程大致是 Objc.getInstance(), 获取 objc 运行时，从运行时生成类对象调用 objc.getClass, 然后在生成的对象上 call 相应的方法，由于 objc 的代码使用的主要是 objc 的对象做输入输出，所以这也是我们增加上述辅助对象的原因，我们使用原生 java 兼容的数据类型作为参数，避免了中途转换的麻烦。

```

MachOLoader loader = (MachOLoader) emulator.getMemory();
loader.setObjcRuntime(true);
Module module = emulator.loadLibrary(new File(path));
//module.callEntry(emulator);
classDumper = ClassDumper.getInstance(emulator);
objC = ObjC.getInstance(emulator);
objcHelper = objC.getClass( className: "ObjcHelper");
keyList = objcHelper.call( selectorName: "NewStringArray::",
    ...args: "functionId,body,openudid,client,clientVersion",",");
jdSignService = objC.getClass( className: "JDSignService");
System.out.println(ObjcClass.create(emulator,keyList).getDescription());

```

接下来就是传入参数，调用相应的 objc 方法获取签名了，

```

public String getSignText(String functionID,String body,String uuid,String client,String clientVersion)
{
    Pointer dic = objcHelper.call( selectorName: "SetDictionaryValue::", ...args: null,"body",body);
    objcHelper.call( selectorName: "SetDictionaryValue::", ...args: dic,"functionId",functionID);
    objcHelper.call( selectorName: "SetDictionaryValue::", ...args: dic,"clientVersion",clientVersion);
    objcHelper.call( selectorName: "SetDictionaryValue::", ...args: dic,"client",client);
    objcHelper.call( selectorName: "SetDictionaryValue::", ...args: dic,"openudid",uuid);
    System.out.println(ObjcClass.create(emulator,dic).getDescription());
    Pointer result = jdSignService.call( selectorName: "getSignWithDic:keys:",dic,keyList);
    ObjcClass objResult = ObjcClass.create(emulator,result);
    String desc = objResult.getDescription();
    String signText = "";
    for ( int i = 0; i< desc.length();i++){
        char c = desc.charAt(i);
        if (c == '{' || c == '\r' || c == '\n' || c == ' ' || c == '}'){
            continue;
        }
        if (c == ';'){
            signText += "&";
        }else{
            signText +=c;
        }
    }
    return signText;
}

```

写完之后执行，果然收到了签名信息，然后立马执行模拟发包，然后，现实又给你沉重的一击，服务器返回签名失败。接下来我们需要使用之前抓取的实际数据来验证 JDSignService::getSignWithDic:keys 的调用结果和进行问题查找，根据静态分析，它是 NSDate 来获取当前时间戳：

```
__mudic = __stubs::__objc_retainAutoreleasedReturnValue();
__stubs::__objc_msgSend(&__OBJC_CLASS_$_NSDate,"date");
dateNow = __stubs::__objc_retainAutoreleasedReturnValue();
__stubs::__objc_msgSend(dateNow,"timeIntervalSince1970");
__stubs::__objc_release(dateNow);
__stubs::__objc_msgSend(&__OBJC_CLASS_$_NSString,"stringWithFormat:",&cf_%.0f);
NSString * __stubs::__objc_retainAutoreleasedReturnValue();
```

NSDate 的实现位于 CoreFoundation，而在 NSDate 里面使用了 c 函数 _gettimeofday 来获取当前时间。

使用 arc4random 来生成 sv 的随机数，

```
__uVar1 = __stubs::__arc4random();
__pNVar8 = (NSString *)
    (ulong)(__uVar1 - (__uVar1 / 3 + ((uint)
__uVar1 = __stubs::__arc4random();
__uVar9 = (ulong)(__uVar1 - (__uVar1 / 3 + ((uint)
__pNVar8 = NSString;
```

因此我们Hook这两个函数，用于访问指定的值，因此能和真机比较返回的数据。

```
Module core = emulator.getMemory().findModule( soName: "CoreFoundation");// _gettimeofday
Symbol gettimeofday = core.findSymbolByName("_gettimeofday");
hookZz.replace(gettimeofday, new ReplaceCallback() {
    Pointer pointer1 = null;
    @Override
    public HookStatus onCall(Emulator<?> emulator, HookContext context, long originFunction) {
        pointer1 = context.getPointerArg( index: 0);
        // String str = pointer.getString(0);
        System.out.println("gettimeofday");
        // context.push(str);
        return HookStatus.RET(emulator, originFunction);
    }
    @Override
    public void postCall(Emulator<?> emulator, HookContext context) {
        // Pointer pointer = context.getPointerArg(0);
        if(pointer1 != null){
            // 这里把 时间写死
            long[] values = {1612947209,07300};
            System.out.println(pointer1.getLong( offset: 0)+"---"+pointer1.getLong( offset: 8));
            pointer1.setLong( offset: 0, value: 1612947209);
            pointer1.setLong( offset: 8, value: 73000);
            //byte[] buf = {(byte)0x91,(byte)0x50,(byte)0xc4,(byte)0x5f,(byte)0x15,(byte)0x97,(byte)0x09,
            //pointer1.write(0,buf,0,8);
        }
    }
});
```

_gettimeofday 的返回值是个结构体，由于这是 ARM64 位，第一个 8 字节是秒数，第二个 8 字节是微秒数。

随机函数固定返回 1

```

IHookZz hookZz = Dobby.getInstance(emulator);
Symbol archRandom = module.findSymbolByName("_arc4random");
hookZz.replace(archRandom, new ReplaceCallback() {
    @Override
    public HookStatus onCall(Emulator<?> emulator, long originFunction) {
        return HookStatus.LR(emulator, returnValue: 1);
    }
});

```

使用的是之前抓的这个测试数据：

```

body---->{"pageNum": "1", "lid": "19,1666,36267,36272"}
functionId---->followV3EnterMainPage
clientVersion---->9.4.0
client---->apple
openudid---->[redacted]
-----keyList-----
functionId
body
openudid
client
clientVersion
[*] Class Name: JDSignService
[*] Method Name: + getSignWithDic:keys:
    [-] Type of return value: object
    [-] Original Return Value: 0x282aec500
st---->1612947209073
sv---->111
sign---->b3b9f901b9fe0bcfca41f24ab5b9a140

```

运行，发现果然是不一样的,确认跑出来的数据不对。

```

gettimeofday
1614311081---736000
gettimeofday 0k
sign=797e4536c1617278d8c0977e9db67f14&st=1612947209073&sv=111&
test offset=7256ms

```

由于执行的结果不对，目前猜测，签名过程中很有可能还有其它数据参与其中，但是虽然值不一样，每次跑出来的却是一样的。为了找到原因，继续静态分析。经过分析，实际干活的函数是这个函数：

JDSignCryptManager::cryptWithPartion:cryptID:content:encrypt:

JDSignService::getSignWithDic:keys 只是把字典转换成表单形式的字符串传给

JDSignCryptManager::cryptWithPartion:cryptID:content:encrypt: 并把返回值做

md5.接下来进入 JDSignCryptManager::cryptWithPartion:cryptID:content:encrypt:

分析。

```
ID JDSignCryptManager::cryptWithPartion:cryptID:content:encrypt:
(undefined8 param_1,SEL param_2,_NSZone *param_3,undefined8 param_4,ID param_5,
int param_6)

{
    int iVar1;
    int iVar2;
    uint uVar3;
    ID iVar4;
    uchar *puVar5;
    undefined8 uVar6;
    uchar *puVar7;
    undefined8 uVar8;
    bool bVar9;

    iVar4 = __stubs::_objc_retain(param_5,param_2,param_3);
    iVar1 = __stubs::_objc_msgSend(param_1,"validatePartion:",param_3);
    iVar2 = __stubs::_objc_msgSend(param_1,"validateCryptID:",param_4);
    if (iVar1 == 0) {
        param_3 = (_NSZone *)0x0;
    }
    if (iVar2 == 0) {
        param_4 = 0;
    }
    iVar1 = __stubs::_objc_msgSend(param_1,"indexFromPartion:cryptID:",param_3,param_4);
    iVar2 = __stubs::_objc_msgSend(param_1,"validatePartion:",iVar1);
    if (iVar2 == 0) {
        iVar1 = 0;
    }
    puVar5 = (uchar *)__stubs::_objc_msgSend(param_1,"keyFromIndex:" iVar1);
    if (param_6 == 0) {
        __stubs::_objc_msgSend(&_OBJC_CLASS_$_NSData,"dataFromBase64String:flags:",iVar4,2);
    }
}
```

这里可以看到这里有取一个 key 的操作，查看 keyFromIndex 函数，发现这里读取一个全局变量：

```
JDSignCryptManager::keyFromIndex:                                XREF[1]:      1066b0c30(*)
    stp                x22,x21,[sp, #local_30]!
    stp                x20,x19,[sp, #local_20]
    stp                x29,x30,[sp, #local_10]
    add                x29,sp,#0x20
    mov                x19,param_3
    adrp                x8,PTR_s_serializeDictionary:_1068ad000     = 1049f59ac
    ldr                param_2=>s_validatePartion:_1049f8c52,[x8, #0x... = "validatePartion:"
                                                                = 1049f8c52
                                                                undefined _objc_msgSend()

    bl                __stubs::_objc_msgSend
    cmp                param_1,#0x0
    csel                param_3,x19,xzr,ne
    adrp                x8,DAT_106b94000                             = ??
    ldr                param_1,[x8, #0xa08]>DAT_106b94a08           = ??
    adrp                x8,UniGCDAsyncSocket.method138.types       = null
    ldr                param_2=>s_objectAtIndexedSubscript:_1049a531a... = 1049a531a
                                                                = "objectAtIndexedSubscript:"
                                                                undefined _objc_msgSend()

    bl                __stubs::_objc_msgSend
    mov                x29,x29
```

点击可以看到写这个值的位置：

DAT_106b94a08

XREF[2]: keyFromIndex::10403252c(R),
FUN_1040325e0:104032658(W)

undefined8 ??

看后面备注, R 是读取的意思, W 是写的位置, 继续分析 FUN_1040325e0 函数, 发现这个函数由 __mod_init_func 引用, 这里需要了解一些 ELF 文件的基本知识, 众所周知, 一个二进制文件会有很多全局变量和静态变量, 这些初始化工作需要在进入二进制的入口点函数之前初始化, 而 __mod_init_func 就是在 ios 上由二进制加载器调用的函数, 那现在问题就明白了, 是因为 unidbg 没有调用这个函数导致导致这个值没有初始化, 查看 unidbg 源码, 发现加载二进制的时候有个参数指定强制初始化:

```
public void Init(String path){  
    Mach0Loader loader = (Mach0Loader) emulator.getMemory();  
    loader.setObjcRuntime(true);  
    Module module = emulator.loadLibrary(new File(path), forceCallInit: true);  
    //module.callEntry(emulator);
```

把参数设置成 true, 运行, 直接提示错误:

Feb 26 12:15:51 unidbg.local JD4iPhone[2155] <Warning>: *** NSForwarding: warning: selector (0x1049a5059) for message 'init' does not match selector known to Objective C runtime (0x10937298e)-- abort

2021-02-26 12:15:51.192 JD4iPhone[2155:3] *** NSForwarding: warning: selector (0x1049a5059) for message 'init' does not match selector known to Objective C runtime (0x10937298e)-- abort

Feb 26 12:15:51 unidbg.local JD4iPhone[2155] <Error>: -[NSAutoreleasePool init]: unrecognized selector sent to instance 0x109418ed0

2021-02-26 12:15:51.830 JD4iPhone[2155:3] -[NSAutoreleasePool init]: unrecognized selector sent to instance 0x109418ed0

猜测是因为 unidbg 现在支持的版本比较低, 没有 NSForwarding 导致的, 那也没关系, 我们直接调用这个初始化函数:

```
module.callFunction(emulator, offset: 0x1040325e0l);
```

这个地址便是 Ghidra 里面看到的这个地址:

```

*****
undefined __cdecl FUN_1040325e0(void)
undefined      w0:1      <RETURN>
undefined8      Stack[-0x10]:8 local_10      XREF[1]:
undefined8      Stack[-0x20]:8 local_20      XREF[1]:
undefined8      Stack[-0x28]:8 local_28      XREF[1]:
undefined8      Stack[-0x30]:8 local_30      XREF[1]:
undefined1[16]   Stack[-0x40]... local_40      XREF[1]:
FUN_1040325e0
1040325e0 f4 4f be a9      stp      x20,x19,[sp, #local_20]!
1040325e4 fd 7b 01 a9      stp      x29,x30,[sp, #local_10]
1040325e8 fd 43 00 91      add      x29,sp,#0x10
1040325ec ff 83 00 d1      sub      sp,sp,#0x20
1040325f0 34 98 00 f0      adrn     x20, got::SDWebImageDownloadImage done

```

接下来再执行程序，可看到已经能够得到正确的结果了：

```

gettimeofday
1614313297---71000
gettimeofday 0k
sign=b3b9f901b9fe0bcfca41f24ab5b9a140&st=1612947209073&sv=111&
test offset=6234ms

```

结尾

经过漫长的分析，我们已经得到了签名的获取方法，而且还是在 PC 上执行的 java 代码，接下来就是模拟发包，搞代理抓手机的 cookie 和其它参数的体力活了，需要提醒的是，JD 的 HTTP 里面有一些不符合 RFC 文档的内容，比如 cookie 里面带了双引号。如果使用那些标准的 HTTP 库来模拟发包就会存在问题。