iOS逆向之抢红包以外的那些事儿

原创 队员编号001 酒仙桥六号部队 4月25日

温馨提示：本文共计4617个字，预计阅读时长12分钟。

1 前言~~

做iOS逆向有段时间了，发现大多数人对于iOS逆向的了解是这样的。

实际上呢，iOS逆向不光只是可以用来简单的开发个插件，用来实现微信抢红包或者钉钉自动打卡功能，它的用途基本包括下面几个大部分：

1. 破解插件

2. 渗透测试

3. 竞对分析

4. 协议破解

接下来让我们依次从上述四个方面展开讲讲iOS逆向。

2 以红包为名浅谈破解插件~~

我们先来说说破解插件，正如字面意思，就是针对某个APP，通过hook的方式修改其运行逻辑，以达到我们的需求，比如通过监听接收微信红包的接口，再调用抢红包这个行为的点击事件，从而来完成一个简单的微信自动抢红包插件。要分析、编写并注入插件，大体思路主要分以下几个步骤：

1. 脱壳：APP在上架AppStore的时候，会统一对其加壳。原理就是一个运行时内存dump的过程。可以用的现成工具也有很多，如frida-dump、Clutch等。

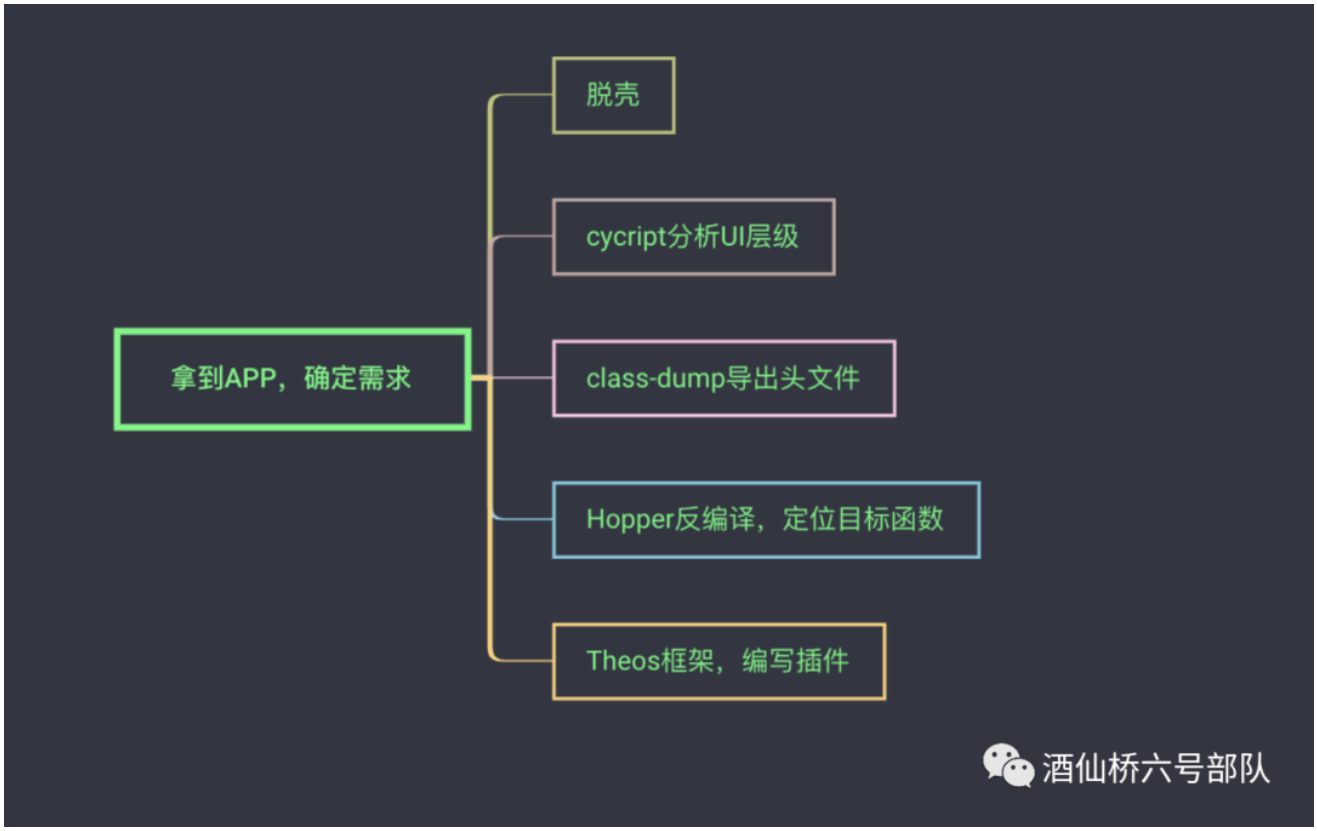
2. cycript：是一个脚本工具，原理就是通过将其注入APP后，然后通过其语法特性将当前或者整个APP的UI层级打印出来。

3. class-dump：用来dump .h文件。实际上就是将所有的类打印出来。

4. Hopper：类似IDA的轻量级反汇编工具，对OC支持性较好。

5. Theos：一个编写逆向插件的框架。

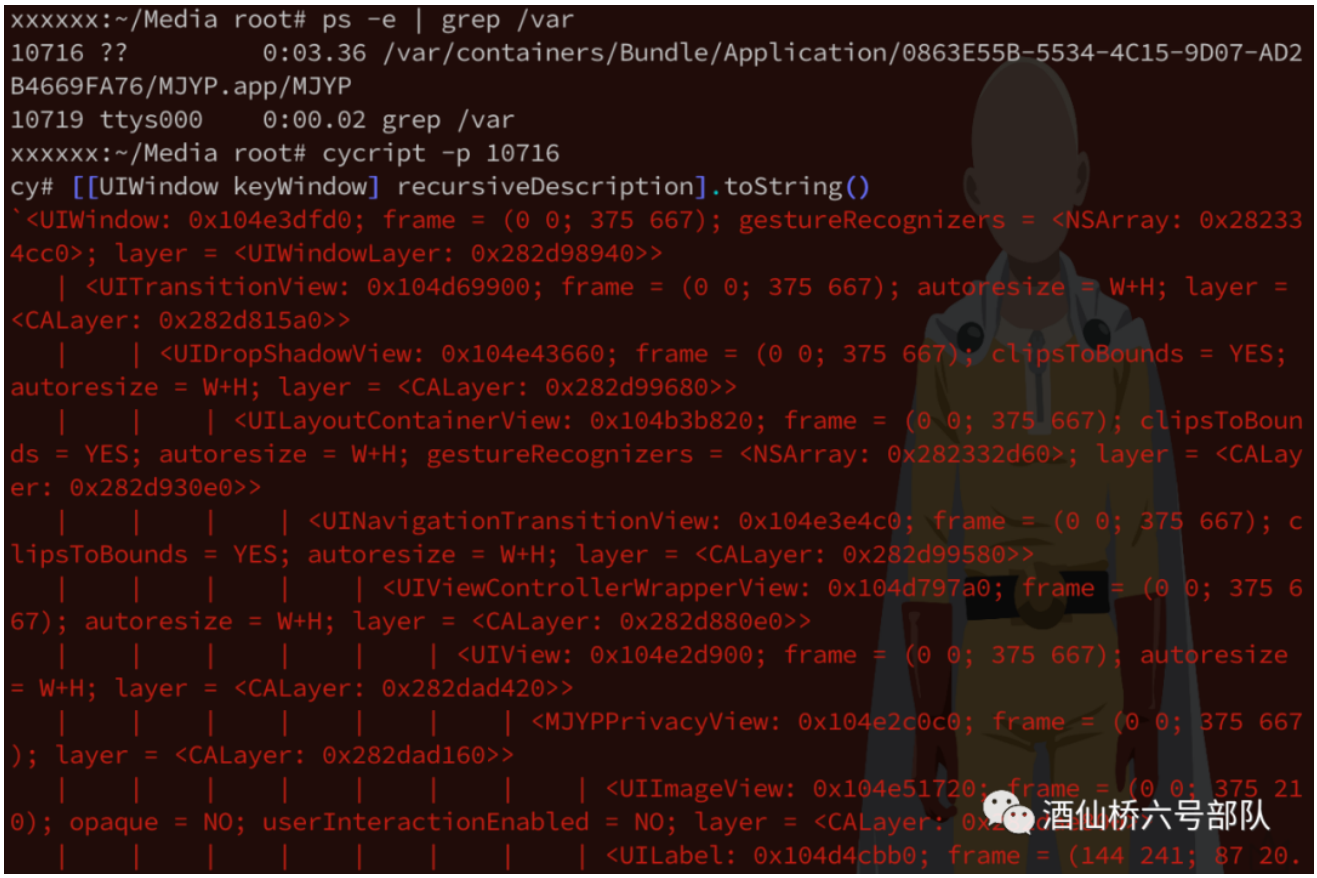
我们平时逆向开发一个插件的大致流程如下：



举例：

拿到一个APP之后，我们通过frida来启动目标进程和dump内存

然后来查看当前APP的UI层级，确定我们目标的类



最后，编写相关代码打包后注入到APP中即可。

但是，这个过程中，我们可能会遇到一个问题，就是反注入；它可能导致我们在分析APPUI层级或者写好插件运行的时候，APP直接闪退。

iOS10以下，反注入基本上是通过在Xcode中设置__RESTRIC字段来忽略加载第三方的dylib

```
static ImageLoader* loadPhase3(const char* path, const char* orgPath, cc
{
  ImageLoader* image = NULL;
  if ( strncmp(path, "@executable_path/", 17) == 0 ) {
    // executable_path cannot be in used in any binary in a setuid proce
    if ( sProcessIsRestricted )
      throwf("unsafe use of @executable_path in %s with restricted binar
  }
  else if ( (strncmp(path, "@loader_path/", 13) == 0) && (context.origir
    // @loader_path cannot be used from the main executable of a setuid
    if ( sProcessIsRestricted && (strcmp(context.origin, sExecPath) == 0
      throwf("unsafe use of @loader_path in %s with restricted binary",
  }
  else if (sProcessIsRestricted && (path[0] != '/' )) {
    throwf("unsafe use of relative rpath %s in %s with restricted binary
  }

```

```
    return loadPhase4(path, orgPath, context, exceptions);
}
```

通过dyld的代码，我们只需要通过010editor修改__RESTRIC的任意某个字母就能实现绕过。

iOS13以后，可以通过下面的环境变量来判断动态库的加载状态来判断是否被注入。

```
char *env = getenv("DYLD_INSERT_LIBRARIES");
```

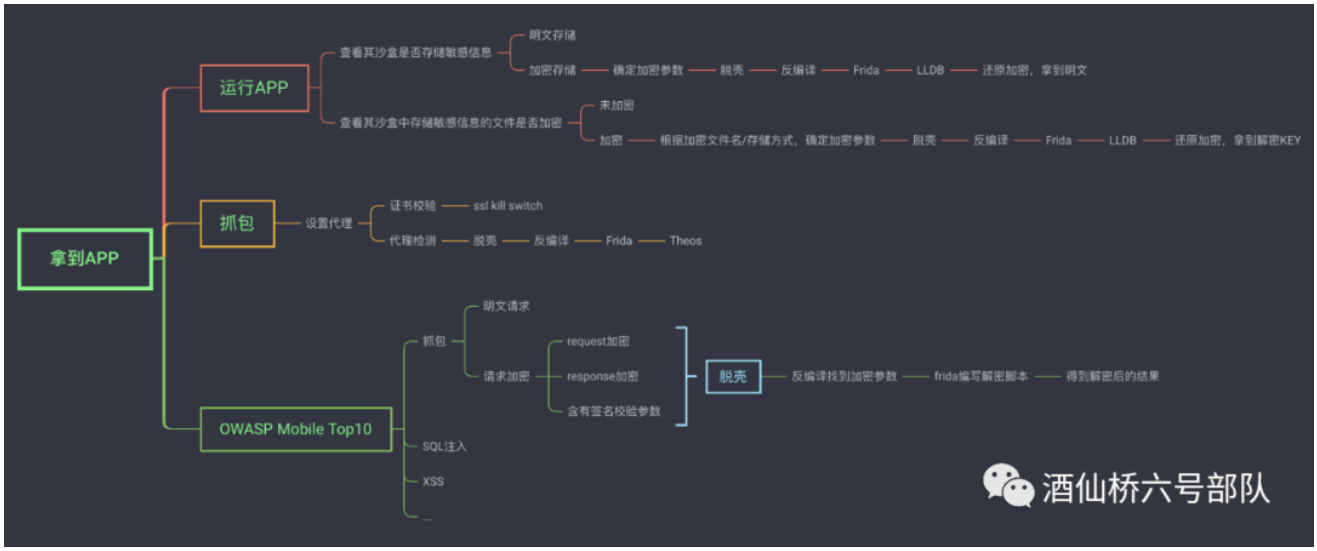
3 以渗透为名浅谈逆向神辅助~~

聊完了各位最关注的破解插件时的一些问题以及思路。我们来接着聊一聊，iOS逆向对于渗透测试的帮助。

通常情况下，我们拿到一个APP，一般会从以下几个方面入手：

1. 运行时的存储
2. OWASP Mobile TOP10

整体流程大概是这样：



运行时的存储很好理解，就是APP在运行时是否在沙盒中存储一些敏感数据，或者一些存储敏感数据的文件，如db文件等是否加密。

OWASP Mobile TOP10这块，可以举个简单的例子。

如果我们在抓包的过程中，遇到一个GET请求，咦，我们想利用一个越权漏洞，但是呢，他加了签名校验参数，这个时候，我们就可以根据上面的流程图，对签名校验参数解密之后再进行测试了。

这里有几个点简单说一说：

1. **抓包，绕过ssl的问题**

我们在抓包的时候，实际上是构造一个中间人，骗服务器和客户端。但iOS系统很多客户端都会校验服务器证书，而ssl killswitch就是patch掉了这个校验过程

2. **越狱检测**

· _dyld_image_count返回dyld映射的当前的image数

· _dyld_get_image_name返回image名称，可以通过它来检索dyld的image名称可以通过检测目录中是否有MobileSubstrate.dylib来确认是否越狱

```
void dylibCheck() {
uint32_t count = _dyld_image_count();
char *substrate = "/Library/MobileSubstrate/MobileSubstrate.dylib";
for(uint32_t i = 0; i < count; i++) {
const char *dyld = _dyld_get_image_name(i);
if (strcmp(dyld,substrate)==0) {
NSLog(@"该设备已越狱");
    }
  }
}
```

绕过也很容易，通过patch相关函数使其return 0 就好。

举例：

下图就是通过检测环境变量加载的dylib来进行越狱检测，我们只需要通过编写插件让这个函数返回0即可。

```
/* @class XHWUtil */
+(bool)checkEnv {
    saved_fp = r29;
    stack[-8] = r30;
    r31 = r31 + 0xfffffffffffffff0;
    r0 = getenv("DYLD_INSERT_LIBRARIES");
    if (r0 != 0x0) {
            if (CPU_FLAGS & NE) {
                    r0 = 0x1;
            }
    }
    return r0;
}
```



```
%hook XHWUtil

+ (bool) checkEnv(

  %orig;
  return 0;

)

%end
```

3.代理检测

代理检测这块，一般是通过CFNetworkCopySystemProxySetting对代理进行判断，大致思路就是：

· 通过反编译，找到与代理函数相关的代码；

· 结合LLDB动态调试，打印其调用栈和寄存器的值，然后做出修改；

· 最后通过Theos框架编写插件即可。

举例：

挂上调试器之后，在CFNetworkCopySystemProxySetting处下断点，跟到下图中后，查看其相关代码逻辑。

所以我们只需要hook相关代码，让其返回0即可。

```
%hook sendDataHttpApi

+ (bool) ewfefweifj(

  %orig;
  return 0;

)

%end
```

4. 反调试

反调试的原理，一句话概述，就是通过一些可用的系统函数如sysctl、ptrace、syscall等，检测到调试器对其的附加后做出相应的处理措施。绕过代码在下面，

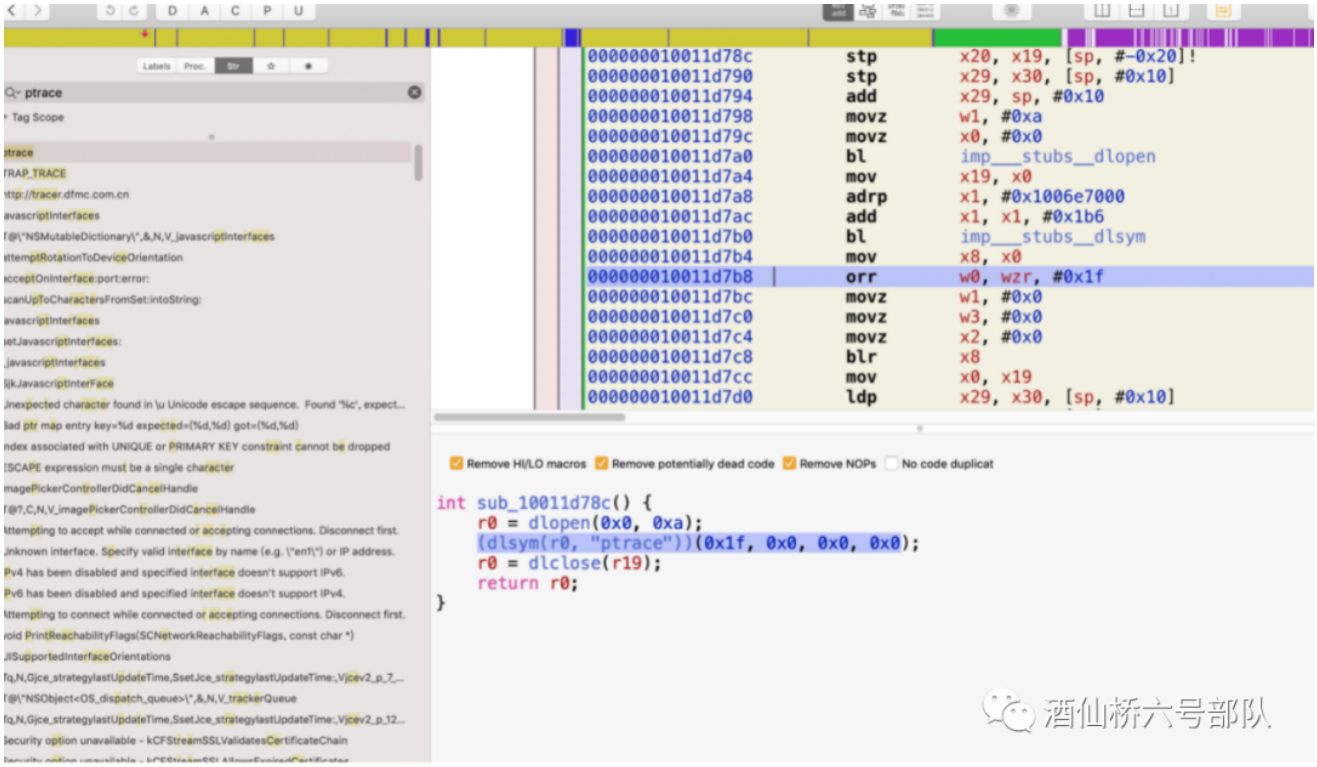
```
#import <substrate.h>
#import <sys/sysctl.h>

static int (*orig_ptrace) (int request, pid_t pid, caddr_t addr, int dat
static int my_ptrace (int request, pid_t pid, caddr_t addr, int data){
  if(request == 31){
    return 0;
  }
  return orig_ptrace(request,pid,addr,data);
}

static void* (*orig_dlsym)(void* handle, const char* symbol);
static void* my_dlsym(void* handle, const char* symbol){
  if(strcmp(symbol, "ptrace") == 0){
    return (void*)my_ptrace;
  }
     return orig_dlsym(handle, symbol);
}
```

举例：

我们先来看看加了反调试后的效果长啥样：

```
xxxxxx:~ root# debugserver *:12345 -a AMapiPhone
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-360.0.26.1
 for arm64.
Attaching to process AMapiPhone...
Segmentation fault: 11
xxxxxx:~ root#
```

再来看看反编译后的代码长啥样：



这种情况下，我们实际上可以直接通过lldb来绕过它。

思路就是：我们通过给ptrace的地址下断点，将寄存器R0的值0x1f改成任意一个值就好。

当然，我们也可以通过编写插件的方式，去绕过它：

```
#import <substrate.h>
#import <mach-o/dyld.h>
#import <dlfcn.h>

static int (*oldptrace)(int request, pid_t pid, caddr_t addr, int data);
static int newptrace(int request, pid_t pid, caddr_t addr, int data){

        return 0;

        if (request == 31) {
    request = -1;
  }
```

```
13    return oldptrace(request,pid,addr,data);
14 }
15 %ctor {
16
17         MSHookFunction((void *)MSFindSymbol(NULL,"_ptrace"), (void *)new
18 }
```

4 浅谈竞对分析~~

聊完了插件破解和渗透测试中我们的一些思路以及遇到的问题，我们来简单聊一聊竞对分析。

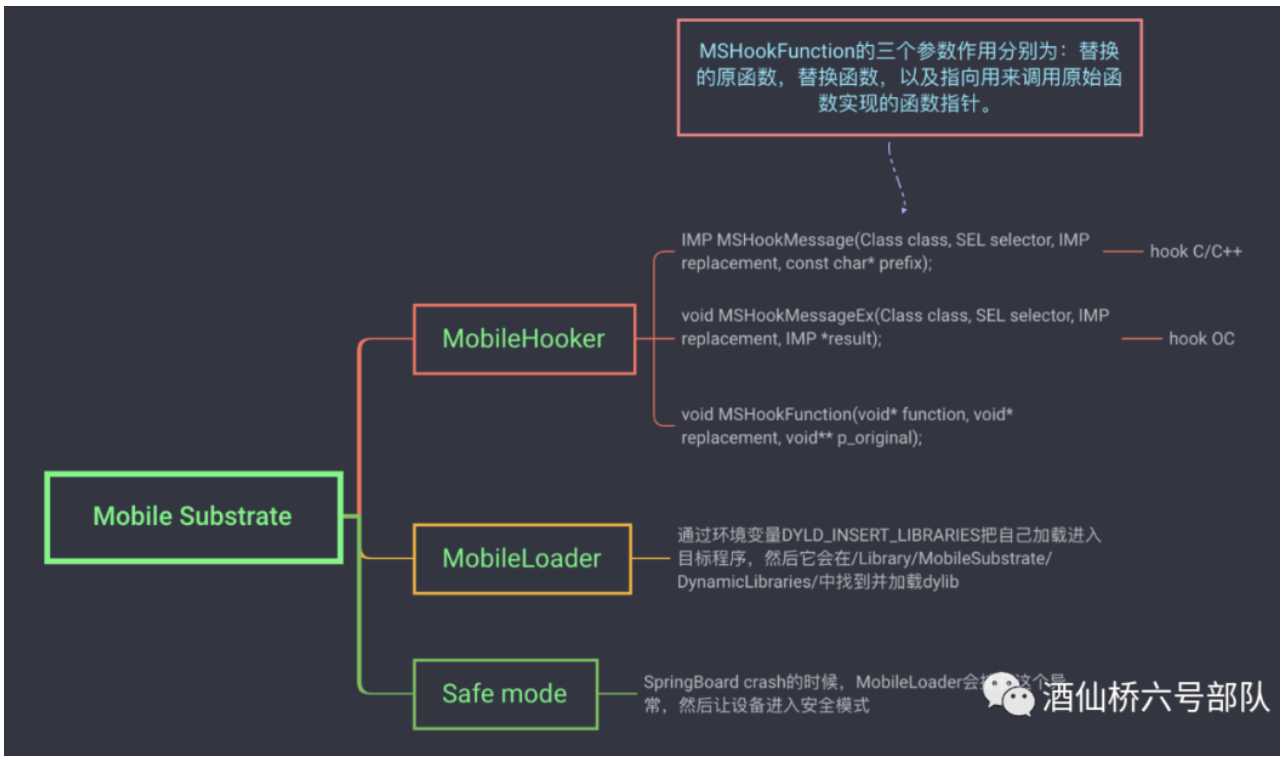
逆向中的竞对分析，不同于常规的带有主观色彩的分析，而是对某个产品、某个功能运行逻辑的逆向解析，结合上面所说的一些绕过手段和调试手段，这里还会涉及到一个代码混淆的东西。APP的关键函数逻辑，一般都分为两种模式编写：

1. C

2. llvm做混淆

如果是用C语言编写函数逻辑，而不去做其他的代码保护措施的话，我们可以通过上述流程，去hookC函数。这里的难点不在于如何hook，而在于怎么确定hook点。

我们平时用来hook的库叫做Mobile Substrate，其中有两个部分，一个是MSHookMessageEx，用于hook OC函数，还有一个MSHookFunction用于hook C或者C++



举例：

我要hook一个sqlite的函数，就用到了MSHookFunctio，代码如下：

```
void(*old_sqlite3_exec)(sqlite3* aaa,const char *sql, int (*callback)(vo

void new_sqlite3_exec(sqlite3* aaa,const char *sql, int (*callback)(voic

  old_sqlite3_exec(aaa,sql,NULL ,NULL ,errmsg);

   NSLog(@"%s",sql);
}

%ctor{
  @autoreleasepool
  {
    MSHookFunction((void *)sqlite3_exec, (void *)&new_sqlite3_exec, (voi
  }
}
```

O-LLVM是基于llvm开发的一个开源项目，目前市面上大多数混淆都是基于这个进行修改的。

OLLVM有三个PASS，分别是：

1. 控制流平坦化：把一些if-else语句，嵌套成do-while语句

2. 指令替换：用效果相同但更复杂的指令序列替换标准二元运算符(+, - , & , | 和^)

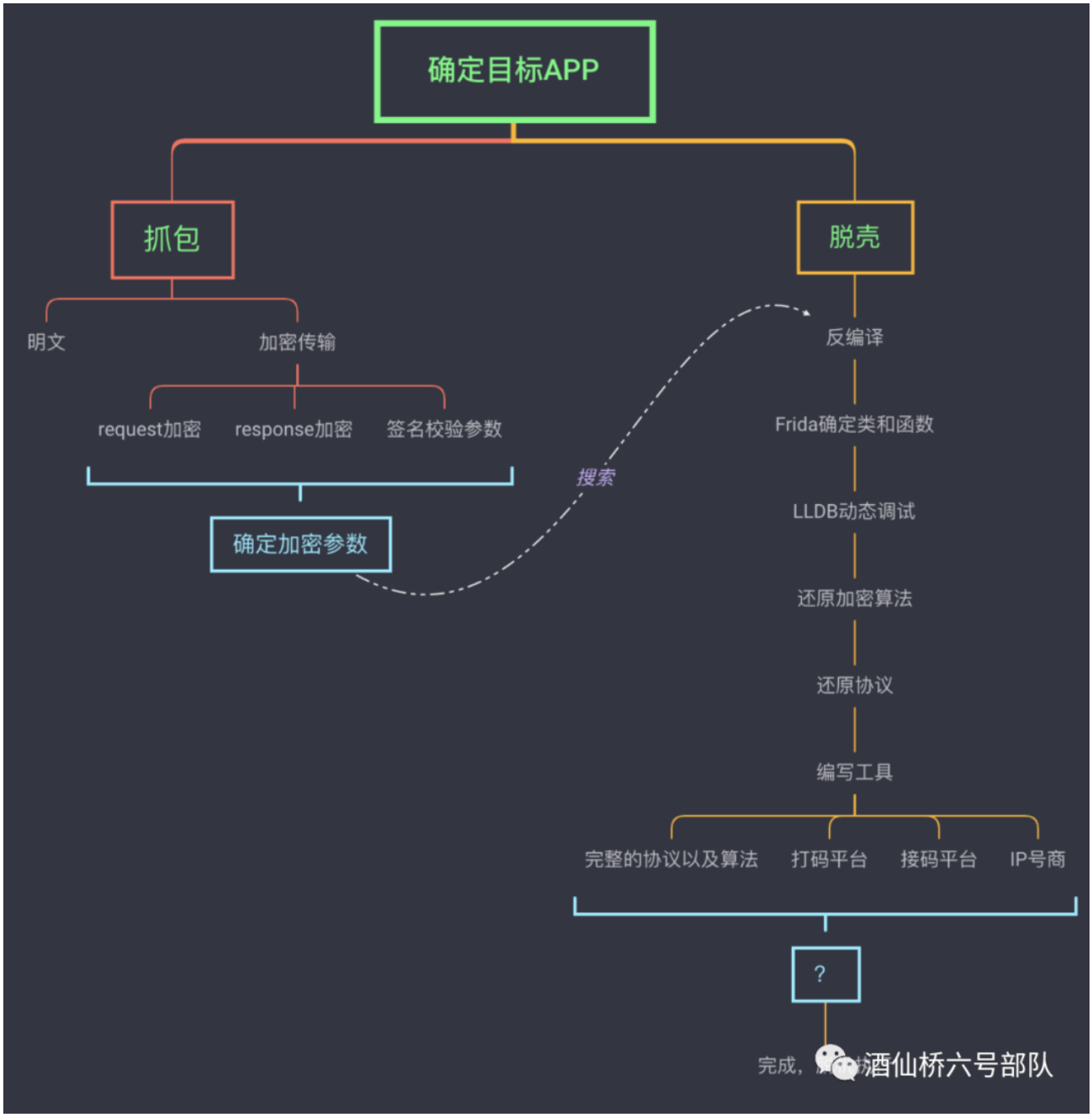
3. 虚假的控制流：主要嵌套几层判断逻辑

LLVM去平坦化这里不细说，我们后面会单独开一篇文章来讲。

5 以灰产为名浅谈协议破解~~

最后我们来讲一下协议破解，其实协议破解，在上文中第二点已经大概讲了一下，目前有大部分用于灰色产业，那么这个产业链到底是怎样的呢？我们可以对其做一个大致分析。

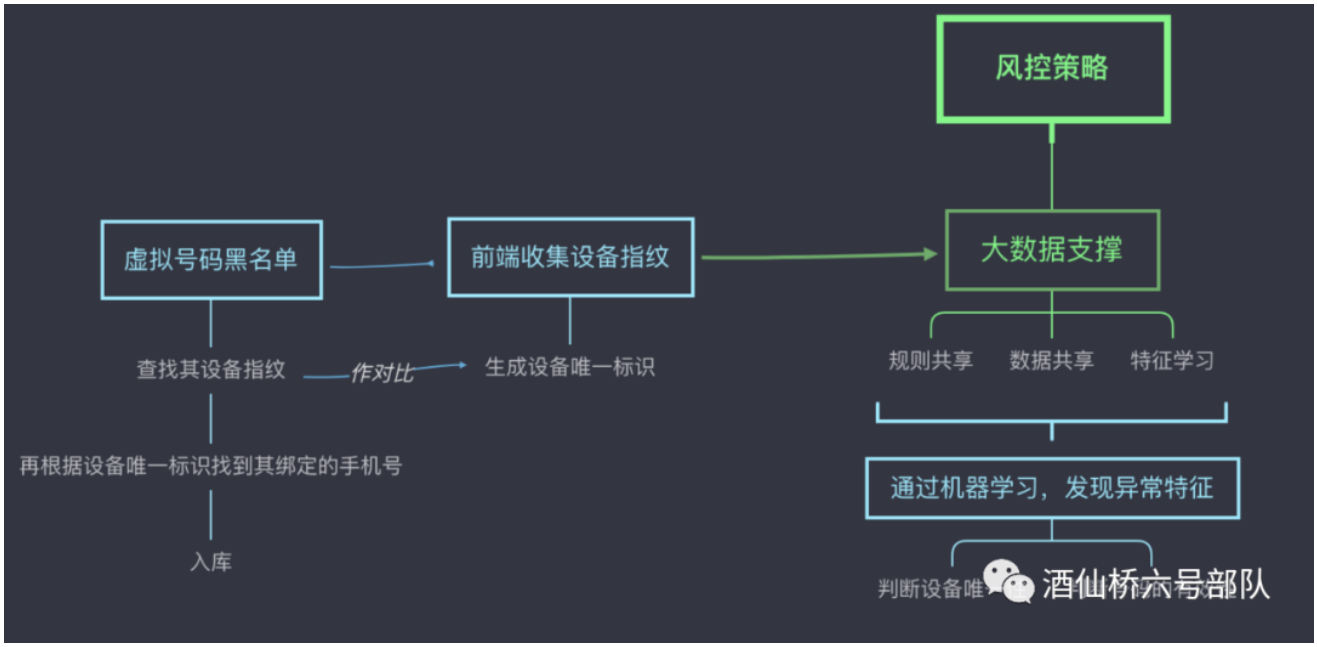
先看一副灰产的简易技术流程图：



上图中有个问号，那么这个就是在做协议分析的时候至关重要的一个东西，如果没有他，即使破解了所有的网络协议，在脱机运行的时候，账号也会被秒封，它就是用来做反欺诈的手段--风控。

那么风控到底是什么东西呢？他又是怎么做到判断设备的唯一性呢？

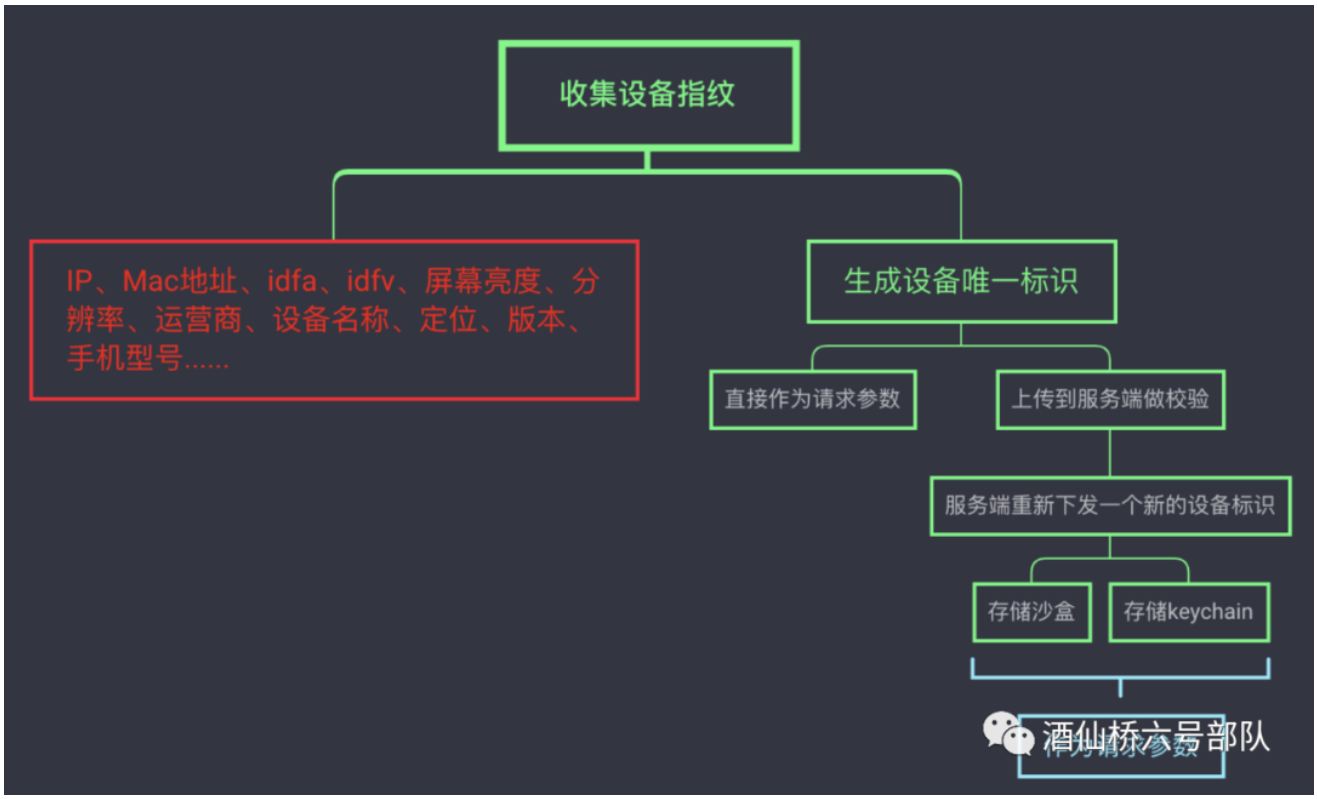
下面用一个粗略的风控策略图来描述下一个最基础的风控策略是怎么来判断设备的唯一性的。



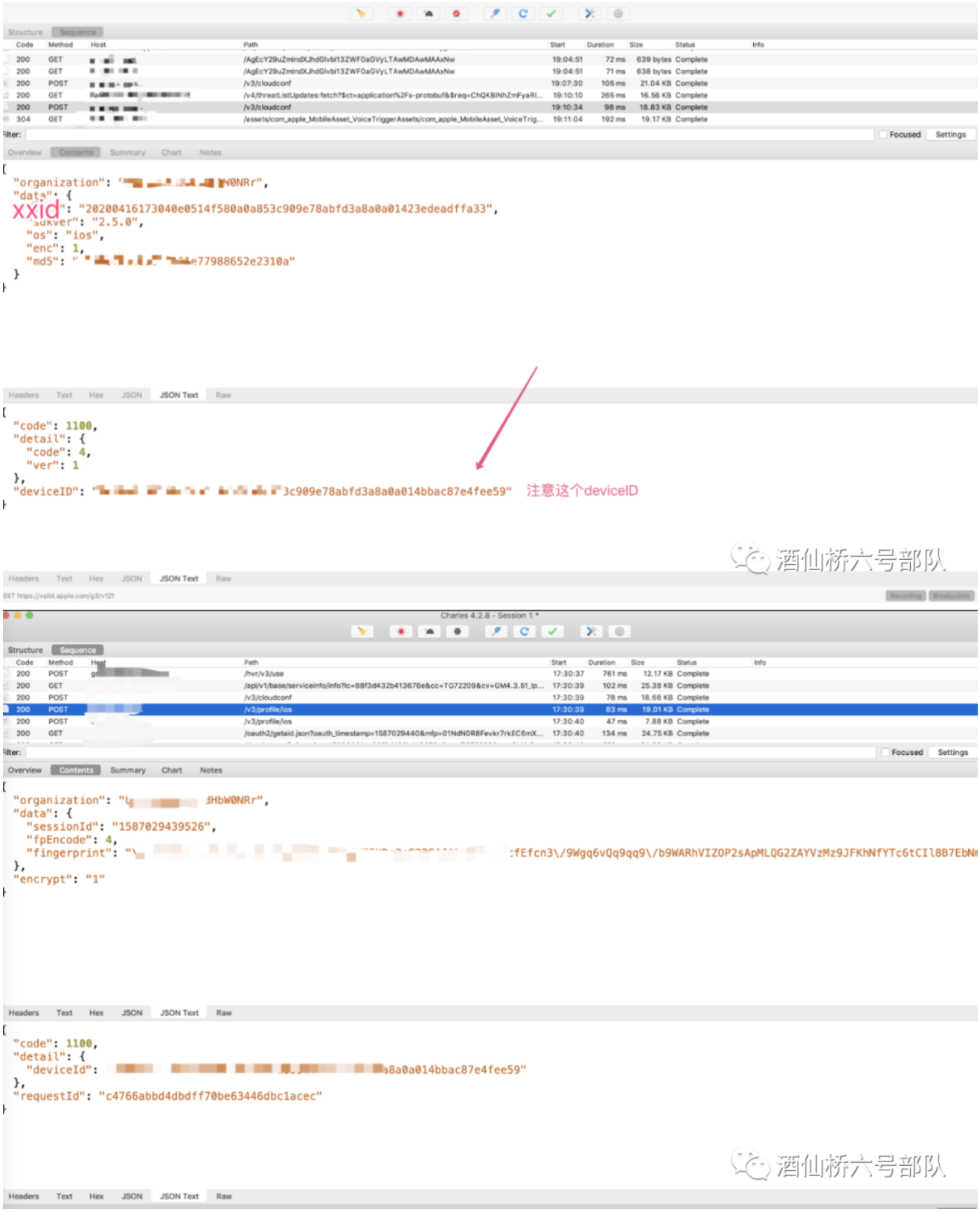
这个图分为两部分：前端和后端。

后端关系到机器学习和数据分析，这里就不赘述了，主要还是看看前端是怎么做的。

通常，APP都是在第一次启动的时候，就已经完成了收集各种设备指纹并且生成唯一的设备ID。



这里举一个简单例子吧：这是一个交友APP，在APP启动的时候，会请求几个风控接口，如下图：



那么在这里，我们可以看到请求中有几个参数要注意：

1. xxid
2. data
3. fingerpring
4. organization
5. deviceid

首先我们先依次看看这几个参数。

1. fingerpring：看名字，能大概猜出来应该是和设备指纹有关，图1的请求中，我们可以看到先通过POST请求提交了一个fingerpring后，服务端返回了deviceID。

2. data：这串值暂时不好确定，待会可以分析看看。

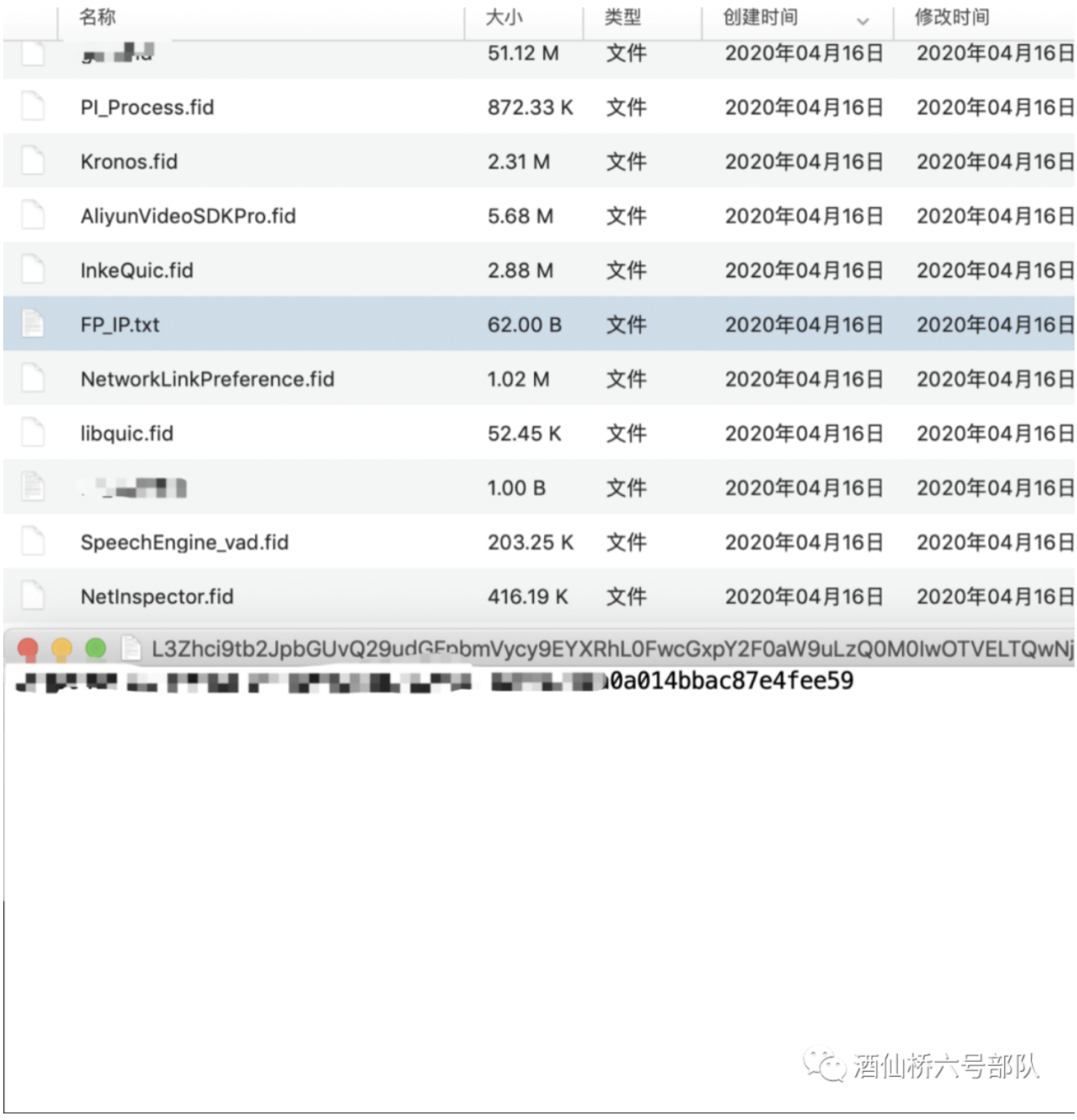
3. xxid：就目前来看，应该是生成的设备唯一标识。

4. organization：暂时不确定，待会分析看看。

5. deviceid：和xxid有点相似，但实际不同。

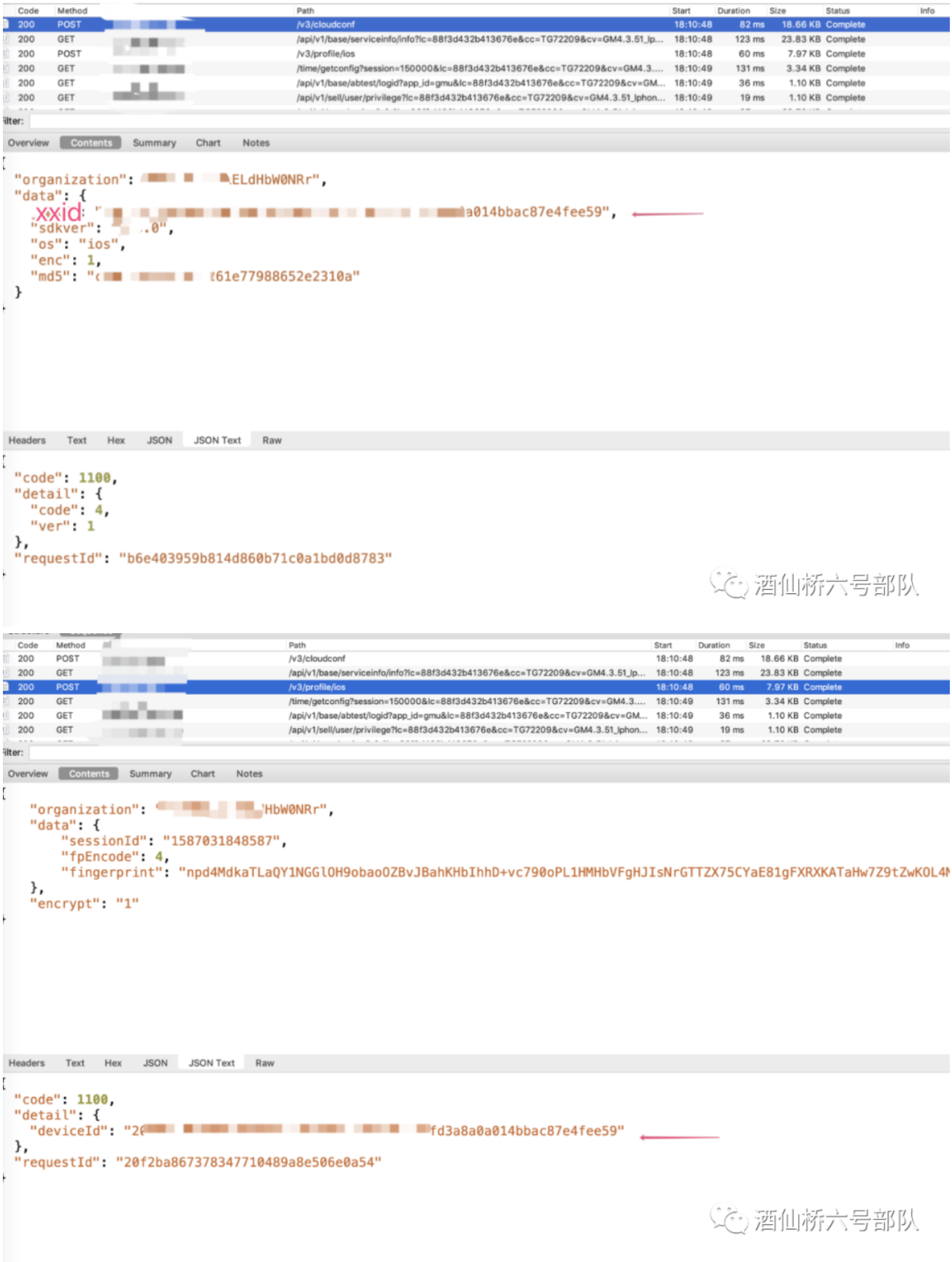
然后，我们就根据目前的这几个参数，结合上文的一些分析流程，来看看具体这几个参数的实现。

第一步，从最简单的做起。为了确定这个APP的设备唯一标识是存储于沙盒还是keychain，我们先在他的沙盒文件中找找，看能不能找到什么有用的信息。



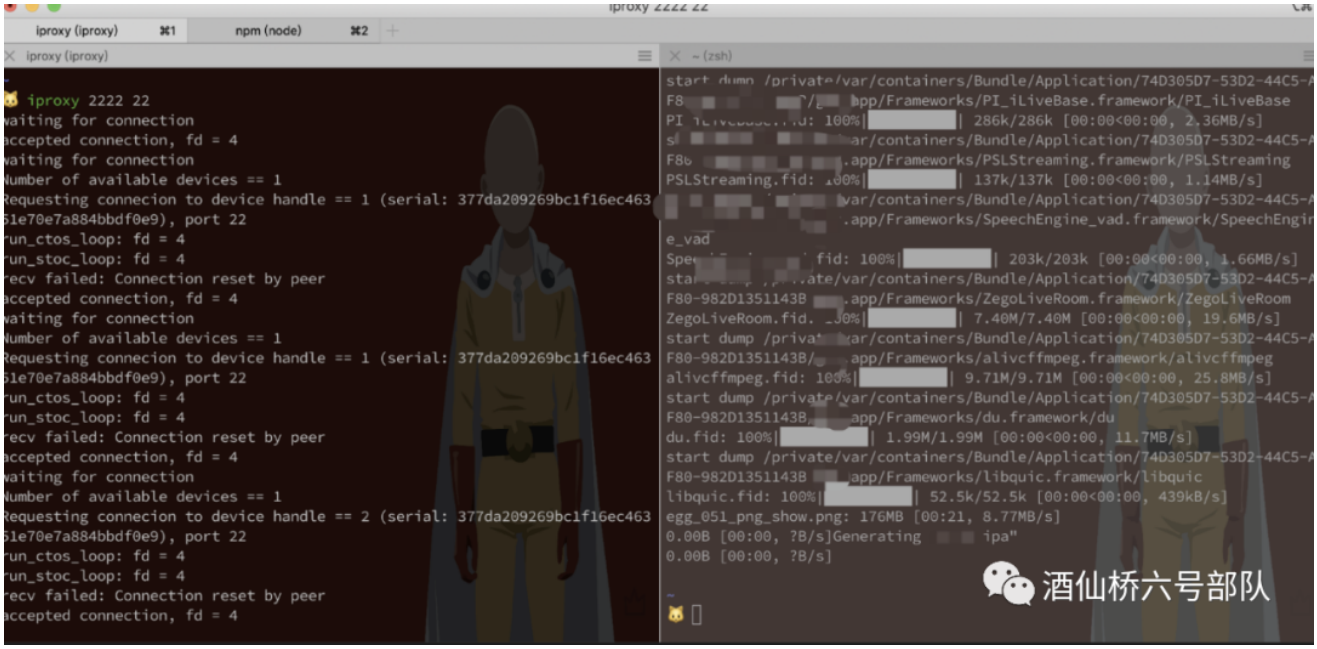
果不其然，我们找到了这样一个文件：FP_IP.txt。打开后可以看到正是我们在网络请求中看到的这串deviceID，在这里我们做一个大胆的假设，后面用到的设备唯一标识应该是这个deviceID。

第二步，为了验证，我们重新启动APP，发现从这次请求开始，无论是xxid还是deviceID,他的值都是第一步中的deviceID的值。和第一次启动APP时的xxid没有任何关系。

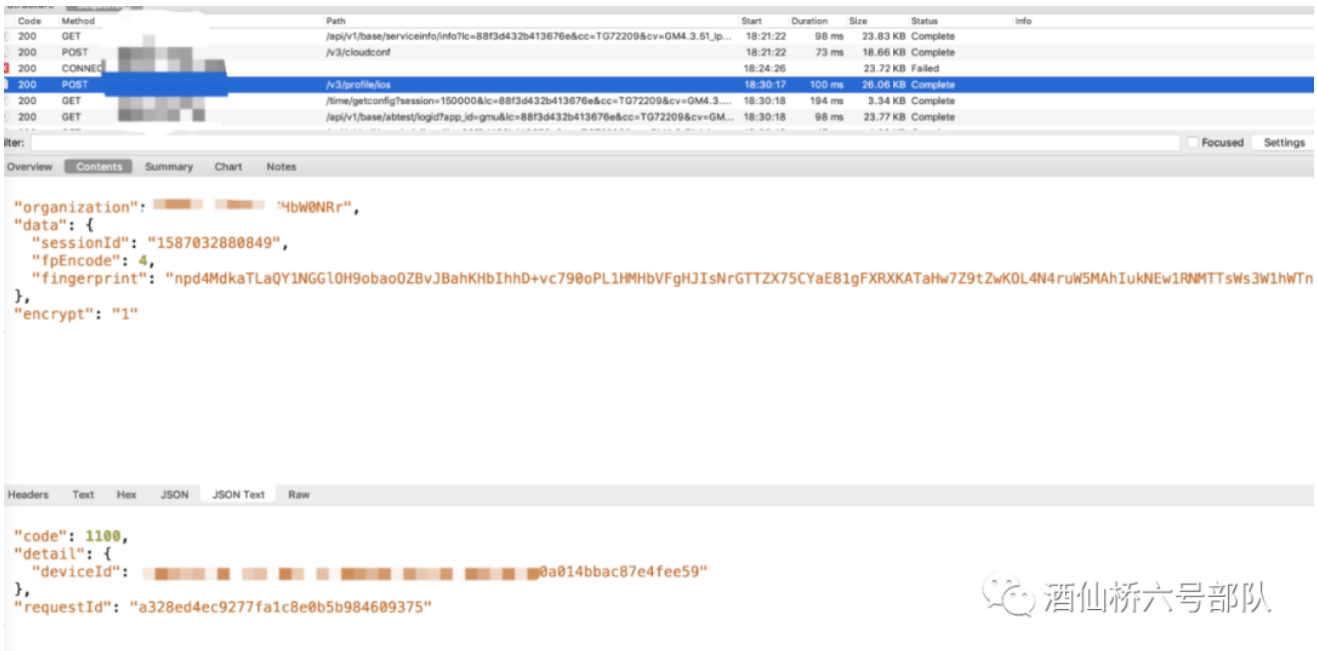


那在这里我们就需要再做一个思考，最开始的那个xxid在这个风控体系里到底起了一个什么作用呢？它到底是怎么生成的呢？这些问题我们留着，看在接下来的分析中会不会有所得。

第三步，我们开始逆向分析。按照前文的流程，先脱壳，再反编译。



这里直接通过frida进行脱壳，原理呢，在文章开头已经讲过了，就是一个简单的内存dump的过程。



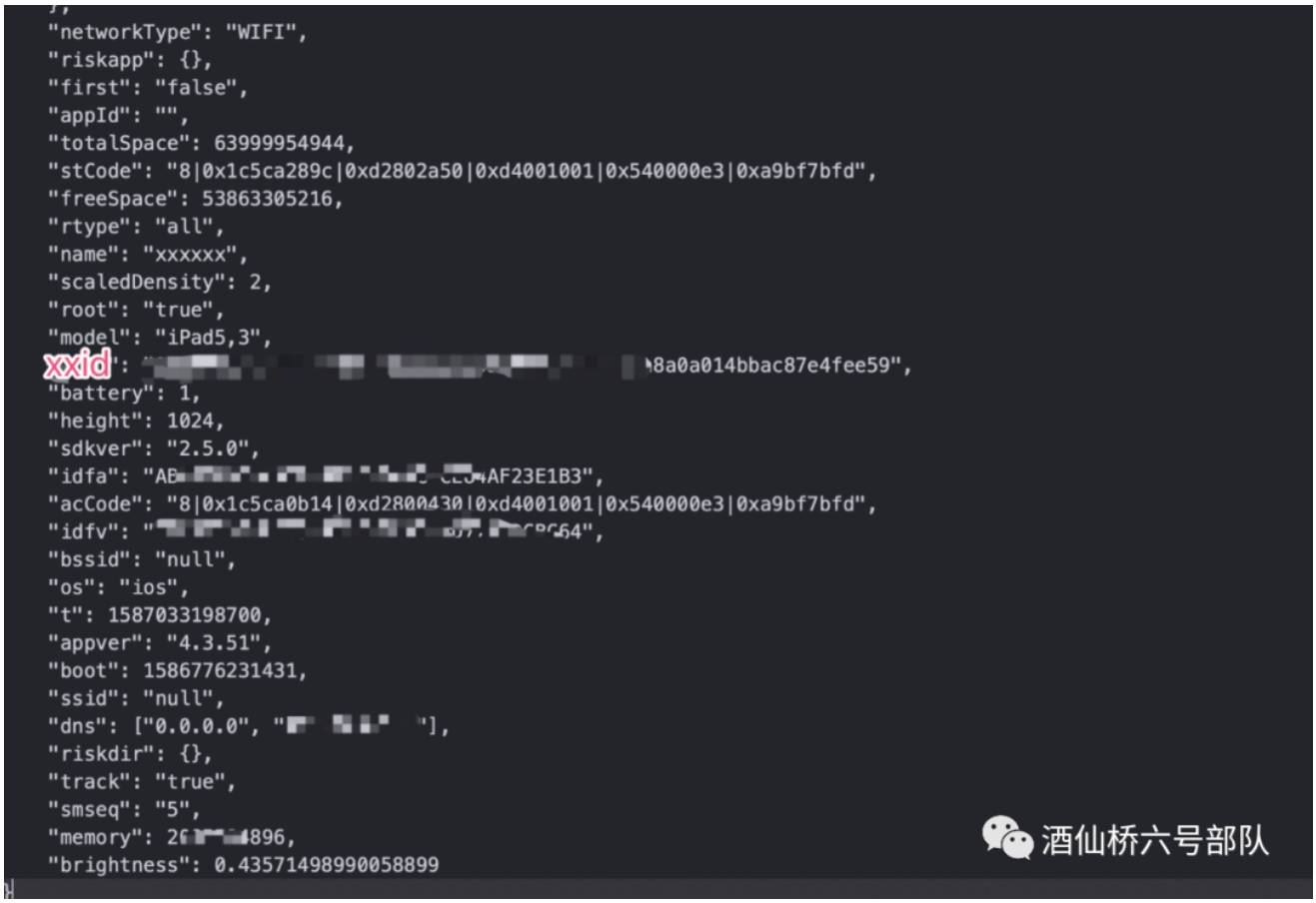
我们通过一系列的抓包、下断点调试等手段，最终确定了fingerpring的加密算法。

(为了保护厂商，我们就不上具体的函数名了QAQ)

```
0814        stur        x0, [x29, var_88]
0818        mov         x0, x2
081c        bl          imp___stubs__objc_msgSend                    ; objc_msgSend
0820        mov         x29, x29
0824        bl          imp___stubs__objc_retainAutoreleasedReturnValue ; objc_retainAutoreleasedReturnV
0828        stur        x0, [x29, var_48]
082c        ldur        x0, [x29, var_88]
0830        bl          imp___stubs__objc_release                    ; objc_release
0834        adrp        x8, #0x102fb8000                             ; 0x102fb8298@PAGE
0838        add         x8, x8, #0x298                               ; 0x102fb8298@PAGEOFF, &@selector(ae
083c        adrp        x9, #0x102fd2000                             ; 0x102fd2dd0@PAGE
0840        add         x9, x9, #0xdd0                               ; 0x102fd2dd0@PAGEOFF, objc_cls_ref_
0844        ldr         x9, [x9]                                     ; objc_cls_ref_SmUtils,_OBJC_CLASS_$
0848        ldur        x2, [x29, var_28]
084c        ldur        x3, [x29, var_48]
0850        ldr         x1, [x8]                                     ; "[illegible]Encryp[illegible]:",@selector(
0854        mov         x0, x9
0858        bl          imp___stubs__objc_msgSend                    ; objc_msgSend
085c        mov         x29, x29
0860        bl          imp___stubs__objc_retainAutoreleasedReturnValue ; objc_retainAutoreleasedReturnV
0864        ldur        x8, [x29, var_38]
0868        stur        x0, [x29, var_38]
086c        mov         x0, x8
0870        bl          imp___stubs__objc_release                    ; objc_release
0874        movz        x8, #0x0
0878        sub         x9, x29, #0x48
087c        orr         w11, wzr, #0x1
0880        sturb       w11, [x29, var_29]
0884        mov         x0, x9
0888        mov         x1, x8
088c        bl          imp___stubs__objc_storeStrong                ; objc_storeStrong
0890        movz        x8, #0x0
0894        sub         x9, x29, #0x40
0898        mov         x0, x9
089c        mov         x1, x8
08a0        bl          imp___stubs__objc_storeStrong                ; objc_storeStrong
```


解密后，我们可以看到很多设备指纹信息。


```
"width": 768,
"sysaddrs": "8|0x1c5ca289c|0x1c5ca13ac|0x1c5c821e4|0x1c5c85ec4|0x1c5ca0e84|0x1c5c86be0|0x1c5aedf7c|0x1c5bb5ce0",
"sysname": "Darwin",
"appname": "[illegible]",
"apputm": "",
"languages": ["zh-Hans-CN"],
"osver": "13.3.1",
"cost": "0[illegible]",
"lstat": {
    [illegible]a0a014bbac87e4fee59": [1, 2]
},
"is_vpn": "false",
"rmCode": "8|0x1c5adb514|0xd102c3ff|0xa9094ff4|0xa90a7bfd|0x910283fd",
"lfrom": "read",
"orientation": "0.000000,0.000000,0.000000",
"s_c": {
    "name": {
        "fname": "\/System\/Library\/PrivateFrameworks\/UIKitCore.framework\/UIKitCore",
        "fbase": "0x1c952b000",
        "sname": "<redacted>",
        "opcode": "8|0x1c9bab81c|0xa9be4ff4|0xa9017bfd|0x910043fd|0x90196088|0x91157d01|0xd01c3082|0x912ea042|0x97008212|
            0xaa1d03fd|0x97010005",
        "saddr": "0x1c9bab81c"
    },
    "mobileNetworkCode": {
        "fname": "\/System\/Library\/Frameworks\/CoreTelephony.framework\/CoreTelephony",
        "fbase": "0x1ca600000",
        "sname": "<redacted>",
        "opcode": "8|0x1ca645b04|0xf9400c00|0xd65f03c0|0x321d07e3|0x1401a1b3|0xf9401000|0xd65f03c0|0x321b03e3|0x1401a1af|
            0x3940a000|0xd65f03c0",
        "saddr": "0x1ca645b04"
    },
    "reachabilityForInternetConnection": {
        "fname": "\/usr\/lib\/libobjc.A.dylib",
        "fbase": "0x1c5bca000",
        "sname": "_objc_msgForward",
        "opcode": "8|0x1c5bcc580|0x90235111|0xf9462a31|0xd61f0220|0xd503201f|0xd503201f|0xd503201f|0xd503201f|0xd503201f|
```



咦，这里也有一个xxid，看这串值实际上也就是我们的deviceID。

第四步，我们又通过一系列的调试，依次找到了xxid的生成算法和organization、md5等的算法。

好了，经过以上的简单分析，我们大致总结出这个风控在前端的一个策略，也刚好回应我们第二步的猜想

这里可能会有朋友会问，为什么还会收集屏幕尺寸，内存、手机名称、型号等看似无用的信息呢?

这就涉及到风控后端的一些策略规则匹配了，总的来说，就是每一个不同的值，都有自己相应的权重，他们的权重大小、占比各不相同，而风控后端又通过各种数据分析和机器学习，总结出一套评分规则，用来判断目标设备是否可信。

所以总结下我们的一个思路应该是:

1. 在确保第一次启动APP的前提下，找到其用来验证设备唯一标识的参数，再通过反编译找到相关函数。

2. 找到我们的目标函数后，在APP启动的瞬间，通过debugserver来附加到进程。

3. lldb下断点，调试。

6 总结~~

以上就是iOS逆向可以在工作中用到的四个方面，我们也逐一举了相关的案例，而在我们的日常安全工作中，iOS逆向还能做更多事，这里只是将平时常见的一些手段列了出来。

作为一个安全研究人员，在熟练运用自身技巧的前提下，还应该尽可能的多做一些思考，比如我们在做iOS端逆向的时候，除了想着怎么去绕过反调试、怎么去除控制流平坦化、怎么样恢复被裁的符号表、怎么样快速找出APP中做的所有安全防护等等，是否应该耗费一定的时间和精力，去做一些能加快自己分析流程的东西？（比如lldb的脚本、frida相关的脚本、iOS应用逆向的时候还应该找到哪些突破点？）

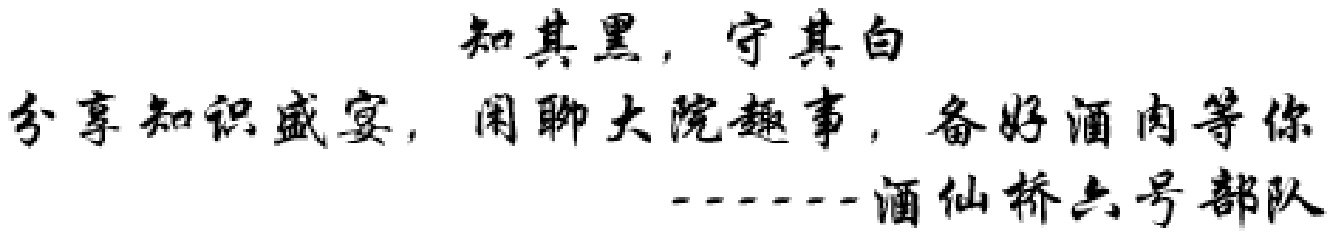
附：本文中用到的环境如下，供君参考

Frida 12.7.22
https://frida.re

HopperDisassembler v4
https://www.hopperapp.com

iOS13.3.1 checkra1n越狱
https://checkra.in

sslkill switch
https://github.com/nabla-c0d3/ssl-kill-switch2



文章已于2020-04-25修改