前言：

现在大厂的设备指纹层出不穷，但是想要确保**稳定性**和**唯一性**高精准其实也挺难的一件事，有的是通过设备信息比重进行的设备ID唯一值确认。比如A设备信息占比10%，B设备信息占比20%，当比重超过60%以上，设备指纹才会发生变化。这样的好处就是当你只修改某一个字段的时候，设备指纹不发生变化。还有的干脆找一个隐蔽的并且唯一的设备信息，作为缓存，每次读取缓存的方式去判断，设备信息是

唯一，比如常见的有Native获取DRM，popen cat /sys/devices/soc0/serial_number ，svc读取bootid并且保存到文件，netlinker获取网卡。都是很常见并且隐蔽的的设备指纹。这篇文章主要介绍了各种指纹的获取情况，如何修改，站在上帝视角去俯看，攻击者和被攻击者遇到的问题 。

看完这篇文章主要你能学到如下：

- 1，常用的指纹检测有哪些？
- 2，如何修改设备指纹，难点在哪里，会有哪些坑需要踩？
- 3，现在的国内的指纹面临哪些问题？

前奏知识：

Android IPC代理是什么？

Android本身是CS架构，客户端（client）服务端（server），我们常用的通过context上下文调用的API都是直接调用代理人的方式去调用的，而真正的服务端是ActivityManagerServer 简称，AMS， 他有很多代理，比如PackageManager，ActivityManager 等，这些都是AMS的**代理人**。而AMS就是**被代理人**。

> 代理模式是一种设计模式，代理人可以提供被代理人的部分或者全部功能，实现代码封装，做鉴权，代码安全的角度，代理模式很常用的设计模式。

AMS和代理们通过Binder进行通讯，Binder是什么，有什么好处这里就不详细展开了，安卓面试八股文，可以理解成进程间通讯的东西，底层实现是通过共享内存，数据传输，读取速度更快。当我们调用代理人的API得时候，本质上是通过Binder去发送一些数据包，和AMS通讯，当AMS收到消息以后把结果在传输给对应的代理人。然后返回给调用方。在每个Manager里面都有一个代理人 。

之前很久之前有一种动态代理的技术，原理就是替换了里面的代理人，因为代理是一个接口，然后我们自己通过Proxy这个类创建一个代理，然后反射set回去，就可以实现常用的API拦截和Hook。类似VA的沙盒，对多开的App提供一份自己实现的代理，然后控制这些代理的返回值，以此实现沙盒相关操作。还有一种比较好的过APK签名的方法就是直接Hook"水管" 也就是hook binder的通讯的方法，当接收到指定事件以后，直接修改具体的结果，以此对Java层进行全量Hook(binder的通讯方法被Hook以后，调用者和代理人只能拿到被修改以后的结果，以此实现Java层的全量Hook，后面再讲签名验证的时候我在详细说。)

如何绕过9.0隐藏API限制？

因为在获取或者分析的时候，需要绕过反射9.0限制，这里采用的是Lsp作者的 AndroidHiddenApiBypass进行隐藏API的调用 ，

在手机装了EDXP或者Magisk，以后一般隐藏API的限制都是默认去除的。因为在Edxp和Magisk里面也需要进行使用一些隐藏API。

设备指纹：

设备指纹主要分为三部分，Java层设备指纹，Native设备指纹，popen执行一些命令获取设备信息，包括一些核心的设备指纹。

这篇文章也主要围绕这三部分进行展开讨论。比如一些内核文件等信息，我也会放在Native层进行讨论。

所有的每个设备指纹我都分为两部分 **Get（如何获取，站在开发者角度） Mock (如何进行修改测试，站在攻击者角度)**

所有的指纹我都分为三种类型，**普通指纹，次要指纹，重要指纹。**

Java层设备指纹：

Setting相关（重要）：

Get:

在setting里面大家经常遇到的可能就是android id的获取的

API如下:

```
Settings.Secure.getString(context.getContentResolver(),Settings.Secure.ANDROID_ID)
```

但是其实Setting里面还有很多别的功能东西，常见的就是Settings.Secure 和 Settings.Global

在Settings.Global 里面其实还有一些别的字段，具体API如下。这些都是一些比较隐蔽的设备指纹。

```
Settings.Global.getString(context.getContentResolver(),"mi_health_id")
Settings.Global.getString(context.getContentResolver(),"gcbooster_uuid")
Settings.Global.getString(context.getContentResolver(),"key_mqs_uuid")
Settings.Global.getString(context.getContentResolver(),"ad_aaid")
```

Mock:

- 方法Hook

Global和Secure 都是实现的NameValueTable接口。

```
public static String getString(ContentResolver resolver, String name) {
       return getStringForUser(resolver, name, resolver.getUserId());
}
```

底层调用的是getStringForUser(resolver, name, resolver.getUserId()) 三个参数，如果Hook的话可以对这个方法进行入手。

Settings.Secure->getStringForUser & Settings.Global ->getStringForUser

- 内存反射：

很多开发者会采用内存反射的方式去获取变量，所以仅仅是通过mock方法的方式不够，如果进行Mock需要将

Settings.Secure 和 Settings.Global 里面的内存变量进行修复，Settings.Global是放了一些全局变量，Settings.Secure放一些安全相关，

Settings.Secure->getStringForUser Settings.Global ->getStringForUser 和 具体方法如下。

```
        private static final HashSet<String> MOVED_TO_GLOBAL;
        private static final NameValueCache sNameValueCache = new
NameValueCache(
                CONTENT_URI,
                CALL_METHOD_GET_SYSTEM,
                CALL_METHOD_PUT_SYSTEM,
                sProviderHolder,
                System.class);

        public static String getStringForUser(ContentResolver resolver, String
name,
                int userHandle) {
            if (MOVED_TO_GLOBAL.contains(name)) {
                Log.w(TAG, "Setting " + name + " has moved from
android.provider.Settings.Secure"
                        + " to android.provider.Settings.Global.");
                return Global.getStringForUser(resolver, name, userHandle);
            }
            ...
            return sNameValueCache.getStringForUser(resolver, name, userHandle);
        }


        @UnsupportedAppUsage
        public static String getStringForUser(ContentResolver resolver, String
name,
                int userHandle) {
            if (MOVED_TO_SECURE.contains(name)) {
                Log.w(TAG, "Setting " + name + " has moved from
android.provider.Settings.System"
                        + " to android.provider.Settings.Secure, returning read-
only value.");
                return Secure.getStringForUser(resolver, name, userHandle);
            }
            if (MOVED_TO_GLOBAL.contains(name) ||
MOVED_TO_SECURE_THEN_GLOBAL.contains(name)) {
                Log.w(TAG, "Setting " + name + " has moved from
android.provider.Settings.System"
                        + " to android.provider.Settings.Global, returning read-
only value.");
                return Global.getStringForUser(resolver, name, userHandle);
            }

            return sNameValueCache.getStringForUser(resolver, name, userHandle);
        }
```

可以看到，**整体的cache都是放在sNameValueCache变量和MOVED_TO_GLOBAL变量内部进行存储**。

我们可以直接反射MOVED_TO_GLOBAL这个HashSet或者去sNameValueCache 这个变量然后去获取这个值的话，也是很容易可以拿到最真实的值的。所以光mock是不够的。比如很多大厂就是**Android高版本绕过了反射限制以后，或者判断当前手机没有API反射限制以后直接通过反射变量的方式去获取。**
sNameValueCache在高版本是一个对象，低版本安卓他是一个ArrayMap 这块需要注意。
sNameValueCache修改的话可以调用API putStringForUser 往里面强制赋值 。这么一来下次对方在通过API去调用的时候就会拿到你已经进行过Mock的值。所以你修改的时候需要进行判断，当前获取的值是否是你已经Mock过的。

蓝牙网卡MAC（普通）：

蓝牙的网卡不是普通的网卡，后面会介绍netlinker获取真实的网卡。

Get:

主要方法就是通过BluetoothAdapter->getAddress

```
    public String getAddress() {
        try {
            return mManagerService.getAddress(mAttributionSource);
        } catch (RemoteException e) {
            Log.e(TAG, "", e);
        }
        return null;
    }
```

Mock:

可以看到这个方法主要是通过IPC的代理类方式去获取的。**所以Hook的话尽可能先Hook代理的IPC类。先尝试反射 android.bluetooth.IBluetooth$Stub$Proxy然后Hook IPC里面的getAddress 而不是直接HookBluetoothAdapter->getAddress**

因为很多大厂获取设备的指纹的时候会检测这个方法是否被Hook，检测也很简单，只需要获取这个artmethod结构体以后

判断这个方法入口是否被替换，比如Sandhook之类的常用的Hook框架，低版本采用的是inlinehook形式，在高版本里面采用的是入口替换，可以直接获取到方法的入口的函数地址，判断一下函数所在的so即可。所以尽可能HookIPC的方法。如果用XPosed去修改的话，还需要注意魔改，否则大厂会通过XposedHelpers->sHookedMethodCallbacks变量把你Hook的方法进行上报。

> 小技巧：这个变量是一个静态变量，所以我们只需要拿到XposedHelpers这个class即可。想要拿到class必须先拿到这个类的classloader，正常的Xposed是通过系统的classloader作为父类classloader，但是edxp这种，是一个方法内部的成员变量，没有任何地方引用这个classloader，所以想拿到这个classloader需要用到内存漫游。把内存全部的classloader都从内存抠出来，然后挨个去反射获取XposedHelpers 即可。代码可参考如下：
>
> https://bbs.pediy.com/thread-269094.htm

sHookedMethodCallbacks里面保存了XPosed全部的Hook方法信息，用于石锤当前方法是否被Hook。获取被Hook方法具体如下：

XposedHelpers->methodCache 不建议使用，如果攻击者使用了XposedBridge->HookAllmethod的话，可能会导致Hook方法上报的遗漏。

```
    private void getHookItemDemo() {
        ArrayList<Object> choose = ChooseUtils.choose(ClassLoader.class, true);
        for(Object obj :choose){
```

```
            ClassLoader clazzloader = (ClassLoader) obj;
            Class<?> clazzXh = null;
            Class<?> clazzBR = null;
            try {
//                clazzXh =
Class.forName("de.robv.android.xposed.XposedHelpers",false,clazzloader);
//                Field methodCacheField =
clazzXh.getDeclaredField("methodCache");
//                methodCacheField.setAccessible(true);
//                HashMap<String, Method> map =(HashMap<String, Method>)
methodCacheField.get(null);
//                for(String key :map.keySet()){
//                    Method method = map.get(key);
//                    Log.e("Test",method.toString());
//                }
                Log.e("Test", "-------------------------------------------------
---------------");

                clazzBR =
Class.forName("de.robv.android.xposed.XposedBridge",false,clazzloader);
                Field callbacksField =
clazzBR.getDeclaredField("sHookedMethodCallbacks");
                callbacksField.setAccessible(true);
                Map<Member, Object> callback = (Map<Member, Object>)
callbacksField.get(null);

                for(Member key :callback.keySet()) {
                    Log.e("Test", "sHookedMethodCallbacks  " + key.toString());
                }
                Log.e("Test", "-------------------------------------------------
---------------");

                return;
            } catch (Throwable e) {
                Log.e("Test","find errror "+e.getMessage());
            }
        }
    }
```

serial（普通）

这个变量在高版本里面基本已经拿不到，及时拿到了也是一个unknow，但是也需要兼容低版本的Android 。

get:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
                return Build.getSerial();
}
```

Mock:

如果返回的不是空，并且不是 unknow 或者UNKNOWN，随机一份原始长度的字符串即可。另外该字段同上，也可以直接对IPC类进行处理，直接Hook IPC对象getSerialForPackage 方法即可 ， 的实现方法具体如下 。

```
public static String getSerial() {
    IDeviceIdentifiersPolicyService service =
IDeviceIdentifiersPolicyService.Stub

.asInterface(ServiceManager.getService(Context.DEVICE_IDENTIFIERS_SERVICE));
    try {
        Application application = ActivityThread.currentApplication();
        String callingPackage = application != null ?
application.getPackageName() : null;
        return service.getSerialForPackage(callingPackage, null);
    } catch (RemoteException e) {
        e.rethrowFromSystemServer();
    }
    return UNKNOWN;
}
```

IMEI , IMSI ,ICCID,Line1Number (普通)

这些基础的Java设备指纹字段没啥好说的，百度一下就能找到具体的获取方法，但是修改的时候需要注意，不要直接Hook，尝试优先Hook ipc即可 。

Build相关（次要）：

Build里面还是有很多有用的东西，比如手机是否开启adb ,usb接口的状态之类的。我们主要将Build里面分为两部分 。指纹相关

又分为两部分，单一字段 ， 复合字段。

1，配置相关

2，指纹相关

- 单一字段（只有一个设备信息）
- 复合字段（多个单一字段复合而成）

这个单独通过Java层去修改是完全不够的，底层走的是system_property_get 这个方法（在native指纹部分会详细介绍），还有要防止popen getprop 这种方法去扫描全部的Build相关参数（popen getprop在popen相关会详细介绍，这里只介绍Java应该如何处理），这个Build相关需要重点关注，他在**Android底层实现类似树状结构。也就是说很多树枝都会有相同的内容。目前所有的作用域一共有七种。** 举个例子，比如常见的fingerprint复合字段系列 ， 就分为如下七种作用域。

> ro.build.fingerprint/ro.build.build.fingerprint/ro.bootimage.build.fingerprint/ro.odm.build.fingerprint/ro.product.build.fingerprint/ro.system_ext.build.fingerprint/ro.system.build.fingerprint/ro.vendor.build.fingerprint

作用域分别如下

```
private static final String Region[] = {
        "build", "bootimage", "odm", "product", "system_ext", "system", "vendor"
};
```

这里面的值构成顺序也都是一样，所以Hook的话也需要全部进行hook，只处理单一是没用的。因为很多大厂做采集，不会只收集一项。

会七个作用域都进行收集。

配置相关：

常见的配置如下，这些字段其实修改不修改不重要，因为很多大厂如果手机开了开发者选项或者debug模式之类的。

会增加当前手机的风险值。所以尝试进行Mock 和修改 。

```
PUT_MOCK_AND_SAVE_ORG("sys.usb.config", "none", null, true);
PUT_MOCK_AND_SAVE_ORG("sys.usb.state", "none", null, true);
PUT_MOCK_AND_SAVE_ORG("persist.sys.usb.config", "none", null, true);
PUT_MOCK_AND_SAVE_ORG("persist.sys.usb.qmmi.func", "none", null, true);
//这两个config可能会拿不到,拿不到则不进行mock
PUT_MOCK_AND_SAVE_ORG("vendor.usb.mimode", "none", null, true);
PUT_MOCK_AND_SAVE_ORG("persist.vendor.usb.config", "none", null, true);
PUT_MOCK_AND_SAVE_ORG("ro.debuggable", "0", null, true);
PUT_MOCK_AND_SAVE_ORG("init.svc.adbd", "stopped", null, true);
PUT_MOCK_AND_SAVE_ORG("ro.secure", "1", null, true);
//手机解锁状态
PUT_MOCK_AND_SAVE_ORG("ro.boot.flash.locked", "1", null, true);
PUT_MOCK_AND_SAVE_ORG("sys.oem_unlock_allowed", "1", null, true);
```

单一字段:

在不修改机型的前提下，下面这些应该都是需要处理的 。大厂扫描频率很高的Build参数 ，随机的话，在原有的基础上开头或者结尾，随机几位数即可。

```
ro.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.build.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.bootimage.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.odm.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.product.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.system_ext.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.system.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.vendor.build.id org-> [RKQ1.200826.002] mock -> [RKQ1.200826.945]
ro.build.version.security_patch org-> [2021-11-01] mock -> [2021-11-19]
ro.boot.vbmeta.digest org->
[ebe54be57a4fb91d8c22c3d69f68651931878d8925eb8a517d8110084fe45513] mock ->
[ebe54be57a4fb91d8c22c3d69f68651931878d8925eb8a517d8110084fa69209]
ro.netflix.bsp_rev org-> [Q8250-19134-1] mock -> [P3660-19134-1]
gsm.version.baseband org-> [MPSS.HI.2.0.c7-00266-
1025_0156_49a7b03461,MPSS.HI.2.0.c7-00266-1025_0156_49a7b03461] mock ->
[MPSS.HI.2.0.c7-00266-1025_0156_49a7b03846,MPSS.HI.2.0.c7-00266-
1025_0156_49a7b03846]

ro.build.date.utc org-> [1639708288] mock -> [1663313901]
ro.build.build.date.utc org-> [1639708288] mock -> [1663313901]
```

```
ro.bootimage.build.date.utc org-> [1639708288] mock -> [1663313901]
ro.odm.build.date.utc org-> [1639708288] mock -> [1663313901]
ro.product.build.date.utc org-> [1639708288] mock -> [1663313901]
ro.system_ext.build.date.utc org-> [1639708288] mock -> [1663313901]
ro.system.build.date.utc org-> [1639708288] mock -> [1663313901]
ro.vendor.build.date.utc org-> [1639708288] mock -> [1663313901]

ro.build.display.id org-> [RKQ1.200826.002 test-keys] mock -> [RKQ1.200826.945]
// maybe debug-key
ro.build.tags org-> [release-keys] mock -> [release-keys]
ro.build.build.tags org-> [release-keys] mock -> [release-keys]
ro.bootimage.build.tags org-> [release-keys] mock -> [release-keys]
ro.odm.build.tags org-> [release-keys] mock -> [release-keys]
ro.product.build.tags org-> [release-keys] mock -> [release-keys]
ro.system_ext.build.tags org-> [release-keys] mock -> [release-keys]
ro.system.build.tags org-> [release-keys] mock -> [release-keys]
ro.vendor.build.tags org-> [release-keys] mock -> [release-keys]
ro.build.host org-> [m1-xm-ota-bd148.bj.idc.xiaomi.com] mock -> [m1-xm-ota-
be811.bj.idc.xiaomi.com]
ro.build.user org-> [builder] mock -> [buizcdn]
ro.config.ringtone org-> [MiRemix.ogg] mock -> [MiRemix.acc]
ro.miui.ui.version.name org-> [V125] mock -> [V635]

ro.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
ro.build.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
ro.bootimage.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
ro.odm.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
ro.product.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
ro.system_ext.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
ro.system.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
ro.vendor.build.version.incremental org-> [V12.5.19.0.RKHCNXM] mock ->
[V12.5.19.0.RKHWCRG]
```

复合字段：

复合字段是多个单一字段拼成的字段，常用的有ro.build.description 还有之前说的7个fingerprint 相关。 这些Mock以后的值要和之前单一字段Mock的值对等。比如某个单一字段值被mock成A 以后，复合字段里面的内也应该是A 。

```
ro.build.description org-> [alioth-user 11 RKQ1.200826.002 V12.5.19.0.RKHCNXM
release-keys] mock -> [alioth-user 11 RKQ1.200826.945 V12.5.19.0.RKHWCRG release-
keys]

ro.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
ro.build.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
ro.bootimage.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
ro.odm.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
ro.product.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
ro.system_ext.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
ro.system.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
ro.vendor.build.fingerprint org->
[Redmi/alioth/alioth:11/RKQ1.200826.002/V12.5.19.0.RKHCNXM:user/release-keys]
mock -> [Redmi/alioth/alioth:11/RKQ1.200826.945/V12.5.19.0.RKHWCRG:user/release-
keys]
```

Get:

```
android.os.SystemProperties->get(key)
```

Mock:

android.os.SystemProperties->get 底层调用的是native_get ， 一个native方法，所以Hook的时候优先处理 native_get

```
android.os.SystemProperties->native_get
```

Java hook完毕以后 还需要反射将Build里面的成员变量进行set。防止采集通过反射的方式去获取

系统默认账号（普通）：

很多大厂会把这个字段也作为指纹的一部分，所以这个方法也需要处理。

Get:

```
AccountManager->getAccounts
```

Mock:

优先Hook ipc

音量相关函数（普通）

Get:

```
AudioManager->getStreamVolume
```

Mock:

优先Hook ipc

传感器相关（普通）：

这个函数不需要太多处理，每个手机类型基本都差不多，每次打乱一下返回结果排序顺序即可。

Get:

```
SensorManager->getFullSensorList
```

Mock:

优先Hook ipc

Java层DRM相关（重要字段）：

这个DRM是水印相关，主要为了处理不同手机加水印的唯一ID 核心的是一个叫deviceUniqueId 的东西，这玩意是一个随机的32位字节数组。很多大厂用这个作为核心的设备指纹，不仅在Java层进行获取，还有在Native层进行获取，在后面Native设备指纹会再次介绍到。

Get:

```
MediaDrm->getPropertyByteArray
MediaDrm->getPropertyString
```

Mock:

Hook的话很简单，这个方法没有IPC底层有自己的实现，直接Hook get的方法即可 。java层Hook是远远不够的，还需要处理native层。

每次随机32位字节数组即可。

Java层网卡信息（普通）：

大厂应该不会信任Java层的mac，底层都是通过netlinker直接获取网卡，或者直接popen执行 ip a 进行网卡信息的全量获取（详细参考后面popen相关介绍）。我直接在底层处理的netlinker socket通讯的时候，所以Java层不进行处理。任何获取网卡的方法，底层最终走的都是netlinker去获取的网卡

> 直接通过netlinker获取网卡，这种方式在安卓10上面貌似已经失效了，但是手机Root以后是没有限制的（亲测android 13 开发板获取成功），这种方式还可以用来检测当前手机是否Root。

但是当执行ip a这种命令的时候，或者调用Java层原始API的时候，底层还是走的netlinker，直接在底层通过ptrace在函数调用执行完毕以后，对寄存器进行Mock 和 Set即可 。详细获取方式可以参考我之前的帖子 。Android netlink&svc 获取 Mac方法深入分析

文件创建时间（次要）：

很多大厂会收集/sdcard/ 或者相册目录的一些创建时间，作为设备指纹，但是很多文件都是默认的1970时间戳，有的少数文件夹创建时间也是很重要的设备标识 。Java里面File对象有文件的创建时间。

Native设备指纹：

聊了挺多Java相关的设备指纹，其实Java层采集的指纹，并不是关键因素，核心的指纹基本都在native层进行处理的。Native部分会详细介绍包括内核文件，还有一些获取指纹的骚操作 。

Build(system_property_get & system_property_read)(重要)

之前在Java层介绍了，Java获取最终总的是native_get，而native_get底层走的就是这个system_property_get 。

在介绍之前我们需要先看看这个函数的源码，android 9以上和9以下实现的方式是不同的。

android 9：

```
__BIONIC_WEAK_FOR_NATIVE_BRIDGE
int __system_property_get(const char* name, char* value) {
  return system_properties.Get(name, value);
}
```

android 9以下：

```
int __system_property_get(const char* name, char* value) {
  const prop_info* pi = __system_property_find(name);

  if (pi != 0) {
    return __system_property_read(pi, nullptr, value);
  } else {
    value[0] = 0;
    return 0;
  }
}
```

安卓9以下是直接实现的这个方法，所以这块又有个细节，android 9 以上 hook __system_property_get 不仅仅需要Hook

入口方法，还需要Hook system_properties.Get 这个方法。

> 很多大厂在android 9以上会直接调用system_properties.Get ， 先解析So获取到system_properties.Get 非导出函数的函数指针，强转成函数指针以后，直接去调用system_properties.Get ， 而非直接调用**system_property_get ， 如果只Hook** system_property_get的话可能就会导致指纹泄漏。所以在android 9以上需要额外处理system_properties.Get(name, value); 这个方法。

如果直接Hook __system_property_get 可能会存在短指令问题。因为这个方法就一个BL指令，普通的inlinehook 可能会失效，这块需要用到异常Hook 。当然也可以直接判断安卓版本号在9.0以上直接Hook system_properties.Get 即可。这个system_properties.Get 是一个非导出函数，需要解析So获取到非导出函数的地址。可以参考sandhook的ELFUtils.cpp 。**同理read方法也是如此，也需要这么处理，在9.0以上需要特殊处理。**

Hook的时候需要注意一件事就是Mock的值长度不能大于原始长度。当 system_property_get 执行完毕以后memcpy 将Mock的value拷贝进去即可 。处理的过程函数如下 。实现也很简单。

```
   int new_system_property_get(const char *name, char *value) {
        int len = origin_system_property_get(name, value);
        string sname = string(name);
        if (gLobalfakeProperties.find(sname) == gLobalfakeProperties.end()) {
            //ALOGE("hook native system_property_get_1 , not replace -> %s value-
> %s " ,name,value);
            return len;
        }
        // get org value
        string fake_value = gLobalfakeProperties[sname];
        if (strlen(value) > 0) {
            memcpy(value, (char *) fake_value.c_str(), strlen(value));
        }
        return len;
    }
```

这块有个细节问题：

为什么不直接在__system_property_find 函数处理?

因为get和read底层走的都是find函数，为什么不直接在find函数处理呢，find函数返回的是prop_info*

这个指针指向的是系统内存的变量，直接写入会直接sign11 如果使用mprotect如果直接对内存变量强制写入可能会导致系统的不稳定，导致出现问题。之前踩过这个坑。所以就只处理了get和find这两个函数。

Get:

使用的话很简单，直接导入头文件就好。

```
#include <sys/system_properties.h>

char sdk[PROP_VALUE_MAX] = {0};
__system_property_get("ro.build.version.sdk", sdk);
```

Native获取DRM ID（重要）

这个指纹也是很多大厂用作唯一ID的核心指纹。处理的话也需要注意，很核心的一个设备指纹ID。

Get:

使用的话很简单，直接导入头文件就好。代码不超过10行 。

导入的头文件实现这个So在mediandk.so里面 ， 所以cmake->target_link_libraries引入的时候别忘记添加mediandk 引入依赖。

这个值不同App 读取的内容都不一样，这块需要注意。

```
#include <media/NdkMediaDrm.h>


const uint8_t uuid[] = {0xed,0xef,0x8b,0xa9,0x79,0xd6,0x4a,0xce,
                            0xa3,0xc8,0x27,0xdc,0xd5,0x1d,0x21,0xed
};
AMediaDrm *mediaDrm = AMediaDrm_createByUUID(uuid);

// 获取 deviceUniqueId
AMediaDrmByteArray aMediaDrmByteArray;
AMediaDrm_getPropertyByteArray(mediaDrm,PROPERTY_DEVICE_UNIQUE_ID,
&aMediaDrmByteArray);
string resut = Base64Utils::Encode((uint8_t
*)aMediaDrmByteArray.ptr,aMediaDrmByteArray.length);
```

Mock:

Hook的话也很简单，直接Hook这个函数地址就行，但是这个方法也是一个短指令，需要用到异常Hook。

处理逻辑如下，因为我们只需要关注description 即可。其他内容不处理。这块有时候直接写入可能会导致问题，需要先mprotect，不能直接用mprotect需要计算一下扇叶大小，是否内存对齐。

```
bool MPROTECT(size_t addr,size_t size,int __prot){
    //计算扇叶
    auto alignment = (unsigned) ((unsigned long long) addr %
sysconf(_SC_PAGESIZE));
    //保证开始地址 减去扇叶大小
    int i = mprotect((void *) (addr - alignment), (size_t) (alignment +
size),__prot);
    if (i == 0) {
        return true;
    }
    LOGE("Helper mprotect error   %s ", strerror(errno))
    return false;
}
```

```
HOOK_DEF_DRM(media_status_t, AMediaDrm_getPropertyByteArray,
             AMediaDrm * drm, const char *propertyName, AMediaDrmByteArray
*propertyValue){
    //LOG(INFO)<< "AMediaDrm_getPropertyByteArray  "<<propertyName ;
    media_status_t array = orig_AMediaDrm_getPropertyByteArray(drm, propertyName,
propertyValue);
```

```
    if(propertyName!=
nullptr&&strstr(propertyName,PROPERTY_DEVICE_UNIQUE_ID)!=nullptr&&
            propertyValue!= nullptr&&propertyValue->ptr!= nullptr&&propertyValue-
>length!=0){
        //LOG(INFO)<< "orig_AMediaDrm_getPropertyByteArray native hook drm start
replcae !";
        //只处理description
        if(gMockdrmid== nullptr){
            LOG(INFO)<< "orig_AMediaDrm_getPropertyByteArray org value == null
!";
            return array;
        }
        MPROTECT((size_t)propertyValue->ptr,propertyValue->length,MEMORY_RWX);
        memcpy((void*)propertyValue->ptr,gMockdrmid, strlen(gMockdrmid));
        LOG(INFO)<< "orig_AMediaDrm_getPropertyByteArray native hook drm  replcae
sucess !";
    }

    return array;
}
```

Netlinker获取网卡信息：

因为之前说过，Linux底层不管什么样的获取网卡，最终底层直接会走Netlinker去获取网卡。在android 10以下可以绕过系统权限从而获取网卡信息，高版本已经失效了 。底层都是svc直接调用recvfrom或者recvmsg去接受socket的消息 。所以不处理svc的话，无法做到全量修改的。**我用的是ptrace 在recvfrom 执行完毕以后，读取参数寄存器，将数据修改以后在重新覆盖寄存器即可 。**处理过程如下。

> 细节点：
>
> socket主要接受消息的函数主要就三个，recvfrom,recvmsg,recv ，netlinker通讯就是通过这三个函数处理的，recv底层调用的是recvfrom ，所以我们只需要处理recvfrom，和 recvmsg 即可。recvfrom执行完毕以后参数是个数组，我们只需要把这个数组buff的值进行覆盖即可，但是recvmsg的话不能这么处理，他的参数是iovec 指针，这个东西大家可以理解成一个箱子。里面装了具体的内容，长度和开始位置 。所以修改的时候需要读取这个开始位置的指针才可以进行set。

处理svc -> recvmsg:

```
void NetlinkMacHandler::netlinkHandler_recmsg(Tracer *tracee) {
    ssize_t bytes_read = TEMP_FAILURE_RETRY(peek_reg(tracee, CURRENT,
SYSARG_RESULT));
    if (bytes_read > 0) {
        // get org value
        word_t msg = peek_reg(tracee, CURRENT, SYSARG_2);
        if(msg == 0){
            return;
        }
        auto *my_Msg = (msghdr *) alloca(sizeof(msghdr));
        int msg_ret = read_data(tracee, (void *) my_Msg, (word_t) msg,
sizeof(msghdr));
        if (msg_ret == 0) {
            //LOGE("ptrace SC_recvmsg msg_ret read sucess ! ")
            auto *iov = (iovec *) alloca(sizeof(iovec));
            int iov_ret = read_data(tracee, (void *) iov, (word_t) my_Msg-
>msg_iov, sizeof(iovec));
```

```
            if (iov_ret == 0) {
                //LOGE("ptrace SC_recvmsg iov read sucess ! buff size ->%lu
",iov->iov_len)
                //保存一份临时的temp buff
                auto *temp_hdr = (nlmsghdr *) alloca(iov->iov_len);
                //temp_hdr使我们自己的创建的,但是在handler_mac_callback_svc 方法里面可
能存在

                int hdr_ret = read_data(tracee, (void *) temp_hdr, (word_t) iov-
>iov_base,
                                        iov->iov_len);
                if (hdr_ret == 0) {
                    //LOGE("ptrace SC_recvmsg hdr read sucess ! ")
                    NetlinkMacHandler::handler_mac_callback_svc(tracee,temp_hdr,
bytes_read);
                    //LOGE("handler_mac_callback_svc sucess ! ")
                    //覆盖和重写
                    write_data(tracee, (word_t) iov->iov_base, temp_hdr, iov-
>iov_len);
                    //LOGE("ptrace SC_recvmsg write_data sucess ! ")
                } else {
                    //LOGE("ptrace SC_recvmsg hdr read  fail ! ")
                }
            } else {
                //LOGE("ptrace SC_recvmsg iov read  fail ! ")
            }
        } else {
            //LOGE("ptrace SC_recvmsg msg_ret read fail ! ")
        }
    }
}
```

处理svc->recvfrom:

```
void NetlinkMacHandler::netlinkHandler_recv(Tracer *tracee) {
    ssize_t bytes_read = TEMP_FAILURE_RETRY(peek_reg(tracee, CURRENT,
SYSARG_RESULT));
    if (bytes_read > 0) {
        //get org value
        word_t buff = peek_reg(tracee, CURRENT, SYSARG_2);
        if(buff == 0){
            return;
        }
        //buff长度
        auto size = (size_t) peek_reg(tracee, CURRENT, SYSARG_3);
        if(size == 0){
            return;
        }
        char tempBuff[size];
        int readStr_ret = read_data(tracee, tempBuff, buff, size);
        if (readStr_ret != 0) {
            LOGE("svc netlink handler read_string error  %s", strerror(errno))
            return;
        }
        auto *hdr = reinterpret_cast<nlmsghdr *>(tempBuff);
        //netlink数据包结构体
        NetlinkMacHandler::handler_mac_callback_svc(tracee,hdr, bytes_read);
```

```
        //将数据写入覆盖掉原来的数据
        write_data(tracee, buff, tempBuff, size);
    }
}
```

netlinker获取的Mac方式详细看我之前的这篇帖子，这里不多重复介绍了 。

Android netlink&svc 获取 Mac方法深入分析

如何使用使用ptrace修改svc可以参考我的另一篇文章。

SVC的TraceHook沙箱的实现&无痕Hook实现思路

上述方法处理完毕以后，哪怕就是执行系统的ip -a命令，拿到的也是被修改以后的值。

内核文件相关（重要）：

内核文件指的是系统的相关文件，很多大厂会直接通过popen cat或者直接fopen只读的方式去读取文件内容。核心的也就那几个。

一般读取的时候都是直接svc openat 底层需要用到svc的IO重定向，如果这块不处理的话，基本没办法进行mock和修改 。

build.prop相关

> "/system/build.prop"
>
> "/odm/etc/build.prop"
>
> "/product/build.prop"
>
> "/vendor/build.prop"

主要就是这四个文件，在低版本这个文件可以直接去读，所以这个文件也需要在mock value以后生成一份新的，作为备份。

生成新的值要和之前Mock的值是一样的，防止出现不相同的情况 。当对方如果使用了SVC读取这个文件的时候，通过SVC的IO重定向绕过读取。

/proc/sys/kernel/random/boot_id

这个ID重启或者刷机以后发生变化，很多大厂会读取这个值，这个值类似一个UUID，SVC读取这个值，然后将这个值保存到私有目录。

跟DRM ID 相比，好处就是不同App读取的值是一样的。一个设备指纹占比很重的值。

/proc/sys/kernel/random/uuid

同上

/sys/block/mmcblk0/device/cid

同上

/sys/devices/soc0/serial_number

同上

/proc/misc

同上

/proc/version

这个是一个linux系统内核文件，里面记录了当前Linux系统版本的相关信息。里面的值类似如下：

```
eg. Linux version 3.18.31-perf-g9b0888a(builder@c3-miui-ota-bd96.bj)
```

这个文件在android 11以上基本读不到了，但是在android 9是可以读到的 。但是android 11有没有什么代替方案呢？答案是有的，svc 调用uname 。 使用方式类似如下，uname也是一个命令行，还可以通过popen uname -a的方式去获取 （popen部分会介绍到）。这个函数在IOS上面也比较实用。

```
    struct utsname buff;
    int i = uname(&buff);
    LOGE("uname sysname %s ", buff.sysname)
    LOGE("uname nodename %s ", buff.nodename)
    LOGE("uname release %s ", buff.release)
    LOGE("uname version %s ", buff.version)
    LOGE("uname machine %s ", buff.machine)
    LOGE("uname domainname %s ", buff.domainname)
```

通过这几项就可以拿到/proc/version 里面的所有信息，**很多大厂会用/ popen uname -a / svc uname 函数 / 和svc openat去读/proc/version以此判断获取的值是否准确，如果有一个对不上都会认为当前设备被修改。**修改方式如下 。一般只需要处理release和Version即可。

```
void handlerLinuxEnvInfo(Tracer* tracee){
        word_t reg = peek_reg(tracee, ORIGINAL, SYSARG_1);
        if(reg == 0){
            LOGI("svc after org peek_reg reg 1 == 0 ");
            return;
        }
        auto *buff = (struct utsname *) alloca(sizeof(struct utsname));
        int msg_ret = read_data(tracee,  buff, reg, sizeof(struct utsname));
        if (msg_ret == 0) {

            string mockLinuxRelease ;
            getZhenxiRuntimeMMKV()-
>getString(LINUX_VERSION_RELEASE,mockLinuxRelease);
            memcpy(buff->release,mockLinuxRelease.c_str() ,SYS_NMLN);

            string mockLinuxVersion ;
            getZhenxiRuntimeMMKV()-
>getString(LINUX_VERSION_VERSION,mockLinuxVersion);
            memcpy(buff->version,mockLinuxVersion.c_str() ,SYS_NMLN);

            //写入内存
            int ret = write_data(tracee, reg, buff, sizeof(struct utsname));
            if(ret!=0){
```

```
                LOGE("linux info write data error %s ", strerror(errno));
            }
        } else{
            LOGE("read_data read data error %s ", strerror(errno));
        }
    }
```

popen相关：

讲之前先简单介绍一下，popen这个函数，这个是Libc里面的导出函数，主要功能是开启一条线程执行某一条shell命令。将返回结果保存到一个临时的File句柄里面，然后直接read即可读取这个文件内容。Android 这个设计我其实有点没看懂，Java层直接执行shell是出异常，但是用popen去执行就没啥问题。

> 注意1：
>
> 因为popen底层走的是execve的这个命令行，是一个shell命令的入口，在64位So里面只要对svc的openat进行IO重定向，哪怕他执行的是execve 也可以进行IO重定向。因为64位execve 底层读取文件，走的也是openat。
>
> 举个例子:
>
> 当我执行popen cat /sys/devices/soc0/serial_number ，如果/sys/devices/soc0/serial_number这个文件被svc openat重定向到/sdcard/a文件。最终cat /sys/devices/soc0/serial_number读取到的也是/sdcard/a ， 而不是原始的/sys/devices/soc0/serial_number 。
>
> 注意2：
>
> popen这个函数它本身会开启一条线程去执行shell ， 因为execve本身就是开一条线程去执行。返回的是一个File 句柄，如果我直接Hook popen 修改他的返回结果，把返回结果替换成我自己的句柄，这样是不被允许的，因为Linux 特性 默认情况A线程的文件创建的文件 B线程无法读取，跨线程找不到文件句柄fd（解决也很简单，可以将文件设置成组内可读即可）
>
> **那么有没有一种方案可以不修改文件权限实现修改返回内容的Mock呢?**
>
> 其实很简单，只需要把参数修改成《cat 被修改的返回结果路径即可》，这样他读取到的内容也是你修改过以后的。
>
> 或者直接吧参数1 设置成“ ”，这样他读取到的内容一定是null。

uname -a

这个命令行在之前介绍过了，主要为了解决/proc/version 读取不到的问题。可以直接通过下面的方式去去获取Linux的一些版本信息。

代码如下，

```
char buffer[100];
FILE *fp3 = popen("uname -a", "r");
if (fp3 != nullptr) {
    while (fgets(buffer, sizeof(buffer), fp3) != nullptr) {
            LOG(INFO) << "uname -a " << buffer;
    }
    pclose(fp3);
}
```

Mock:

这个uname底层走的还是svc的 uname函数，所以修改的时候只需要在svc层面直接修改uname函数返回结果即可。

参考native /proc/version 修改。

getprop

这个执行的内容返回的值和，adb shell 以后执行getprop 结果是一样的。输出的是当前手机全部的Build相关配置。获取代码具体如下 。

```
pfile = popen("getprop", "r");
pfile = popen("getprop | grep dalvik", "r");
pfile = popen("getprop ro.odm.build.id", "r");

while (fgets(buf, sizeof(buf), pfile)) {
    LOGE("getprop -> %s", buf);
}
```

返回结果就不展示了，自己用手机 adb shell 在执行getprop 即可 。

Mock:

Hook的话也很简单，直接Hook popen 提前生产一份已经Mock好的 ,生产的这个要和Java层Build mock的值是一样的 ， 然后直接换成cat 成你自己的文件即可 。这块需要注意，就是 getprop 有三种模式。代码如下

> getprop ro.odm.build.id
> getprop | grep dalvik
> getprop

需要对这种过滤模式进行处理。

内核相关文件的cat:

之前在native层说的内核文件都可以通过popen去cat 。代码如下，修改的话直接svc openat io重定向就好 。

```
FILE *fp = popen("cat /sys/devices/soc0/serial_number", "r");
...
```

ps -ef & ps

这个文件是扫描当前进程的，可以用来做反调试检测，比如刚启动的时候去获取一下当前进程列表。

就可以知道是否存在frida，或者当前进程是否被ptrace ，因为用ptrace调试的话是需要多开启一条调试线程的。

```
pfile = popen("ps -ef", "r");

while (fgets(buf, sizeof(buf), pfile)) {
    LOGE("ps -ef  -> %s", buf);
}
```

返回结果类似如下：

```
2022-07-21 15:33:13.411 2812-2812/包名 I/Zhenxi: instrumentation UID
PID   PPID C STIME TTY          TIME CMD
2022-07-21 15:33:13.413 2812-2812/包名 I/Zhenxi: instrumentation u0_a484
2812    783 19 15:33:12 ?    00:00:00 包名
2022-07-21 15:33:13.413 2812-2812/包名 I/Zhenxi: instrumentation u0_a484
2863   2812 0 15:33:12 ?     00:00:00 ps -ef
```

ip a（重要）

```
pfile = popen("ip a", "r");

while (fgets(buf, sizeof(buf), pfile)) {
    LOGE("ps -ef  -> %s", buf);
}
```

这个也是很核心的设备指纹，里面会获取当前手机的网卡信息，whan0 wlan1 p2p0 这些信息。这个底层走的也是netlinker

所以在netlinker层直接修改拦截，他哪怕执行的命令行也是生效的 。返回的东西很多，可以自己尝试打印一下。很多大厂也会用这种方式去扫描你得网卡Mac地址 。

ls -al /sdcard/Android/data

扫描私有目录，返回私有目录的一些信息 。可以判断当前App是否存在其他App目录下，主要用于检测沙箱。

其实检测沙箱还有一个很好的办法，就是检测手机的进程信息 。如果当前App在自己正常情况启动，只会有一条线程。

但是如果放在VA沙盒内部的话，VA沙盒本身会启动一条线程，自己的App本身也会启动一条线程。所以线程数量就对不上。也可以认为作弊。代码参考如下，**绕过的话也很简单 Hook readdir 当发现读取的是调试线程直接return null即可 。**

```
void Test::checkSandbox() {
    DIR *pdr = opendir("/proc");
    if (pdr == nullptr) {
        return;
```

```
    }
    dirent *read_ptr;
    //在app启动之前检测当前app所有的进程,判断是否存在和main不一样的进程
    while ((read_ptr = readdir(pdr)) != nullptr) {
        int procpid = atoi(read_ptr->d_name);
        LOG(INFO) << "find /proc/ child dir  " << procpid;
        //打开成功&&发现一条不等于主进程id的pid
        if (procpid && procpid != getpid()) {
            //这个文件所属主线程
            LOG(ERROR) << ">>>>>  FIND OTHER THREAD SANDBOX " << procpid;
            closedir(pdr);
        }
    }
    closedir(pdr);
    LOG(ERROR) << ">>>>> NOT FIND SANDBOX ";
}
```

popen扫描Magisk

这些命令都可以进行magisk的列表的扫描，判断当前线程是否存在magisk等关键字，都是很好的办法。

```
popen("df | grep /sbin/.magisk", "r");
popen("mount  | grep /sbin/.magisk", "r");
popen("ps | grep magisk", "r");
```

修改的话也很简单，如果是ps 或者 df 直接生成一份不存在magisk关键字的文件，（还有一些痕迹关键字，比如xposed,edxp,riru这些都是常用的检测关键字）

mout直接 svc IO重定向绕过即可 。

popen logcat

有很多大厂，他当发现你设备信息异常的时候，会直接执行popen logcat 直接扫描你当前手机的日志系统 。

把异常的log都进行上报，用于石锤当前用户是否作弊 。所以这个也需要处理 。代码如下：

```
pfile = popen("/system/bin/logcat -b main -d -v threadtime -t 200 --pid 当前线程
pid", "r");

while (fgets(buf, sizeof(buf), pfile)) {
    LOGE("logcat  -> %s", buf);
}
```

APK签名：

前段时间有人问我如何过Apk签名 。其实现在过APK 签名 非常简单，目前主要的获取签名就两种办法，比较常用，一般大厂和加固都在使用。这里顺手写一下原理 。

Java 层直接 通过binder和 AMS通讯获取真实签名信息。

这种方法的好处我就不多说了，直接和AMS通讯，获取最真实的签名信息。这么一来你不管你Hook pms里面的哪些方法也没啥用。

代码我就不外发了，因为很多大厂和加壳都在用 。在我的课程里面都会讲到 。可以绕过大多数签名的Hook ， 动态代理pms ，

hook 获取签名的方法，用这种方式都可以直接绕过。那有没有什么办法可以绕过这种检测呢?

绕过原理:

因为和AMS通讯需要用到Binder，Binder可以理解成"水管" ，他虽然和AMS直接进行通讯，但是还是要经过我们的水管，我们直接对这个水管处理即可 。在通讯时候对水管进行拦截。在BinderProxy -> transact 的方法里面进行拦截和替换签名信息即可 。

Native层 svc读取/data/app/包名/base.apk 解析zip 解析里面的签名文件信息。

Java层是通过AMS通讯获取签名信息，我们直接用svc openat读取apk文件， 进行手动解析apk 的签名 。不信任系统Api的解析结果。这个也是常用的检测签名办法。这样拿到的结果就是可信的结果。

绕过原理:

svc openat的IO重定向，当他读取原始/data/app/包名/base.apk 的时候我们将它修改成原始apk的路径。

这么一来他读取到的是原始apk路径，而不是被修改的路径，得到的签名也就是原始的签名。底层在处理一下svc readlink readlintat 防止检测路径被替换。

用这两种方案可以目前干掉市面上99%签名检测 。

文外言:

我个人感觉现在的指纹其实都不是很安全，调用的系统API，个人感觉得到的内容都不是值得被信任的。

等我的下一篇文章，如何打造一款直接通过Binder和直接和AMS通讯，手动解析Binder数据包 ，配合XML强制设置启动ISO进程，

> 什么是ISO进程:
>
> android:isolatedProcess="true"
>
> 在启动服务的时候配置一下这个即可，此时这条服务，只能能绑定的binkder进行通讯，不受其他线程干扰。防止初始化时候被检测。或者在Zygote 进行进行初始化 ，使用也很简单配置一下android:useAppZygote="true"即可 。具体使用说明可以google官网看看

并且防止进程被干扰。之前的都是通过系统提供的IPC类进行和AMS通讯，我们可以自己去构造一个替换掉系统的，然后手动和AMS请求通讯，从而实现系统指纹直接获取。这种方案国内我看了一下还没有相关文章 。