“迷惑行为”大揭秘_酒仙桥六号部队 - MdEditor

> “迷惑行为”大揭秘

这是 酒仙桥六号部队 的第 120 篇文章。

全文共计 2289 个字，预计阅读时长 7 分钟 。

前言

这是一个挖掘客户端逻辑漏洞的进阶技巧分享。对于客户端实现了 OTP 加密的情况下。如何进行漏洞挖掘以及利用。

背景

受某银行的委托，我们对其企业版手机银行进行了安全测试。其实 APP 渗透测试相对于 WEB 的渗透测试来说，APP 更具有挑战性，尤其金融类 APP 挑战更大。首先要绕过证书校验，其次在有壳的情况下首先要进行砸壳，砸壳还不一定成功，最后要突破加密算法。然后才能正常进行测试。如果其中任何一个环节没有突破，测试过程将会带来巨大的挑战。

由于金融监管机构对金融 APP 的强监管，制定了一系列针对金融 APP 的安全规范。如：双向证书校验、XXX 方

式加壳，OTP 方式加密，这些都是规范里必须要做的。目前行业里使用双向证书校验的 APP 还比较少。OTP 加密虽然越来越规范，但还是存在一定的问题。

渗透目标

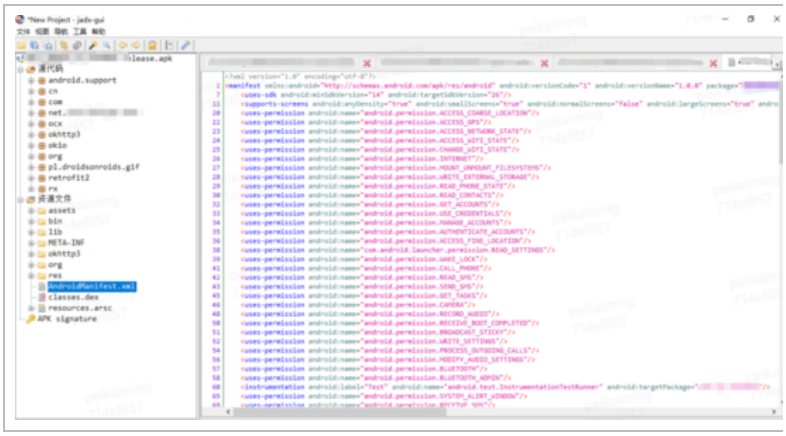
首先我们在测试手机上安装需要渗透的某行 APP, 使用抓包工具进行抓包，结果发现存在证书校验，但是这也难不倒我们，在网上现在又很多的证书校验绕过插件，这里尝试使用 justTrustMe 进行绕过，发现能够成功绕过。

说明他们的证书校验机制并不安全，至于绕过的具体的细节这里就不细说了。

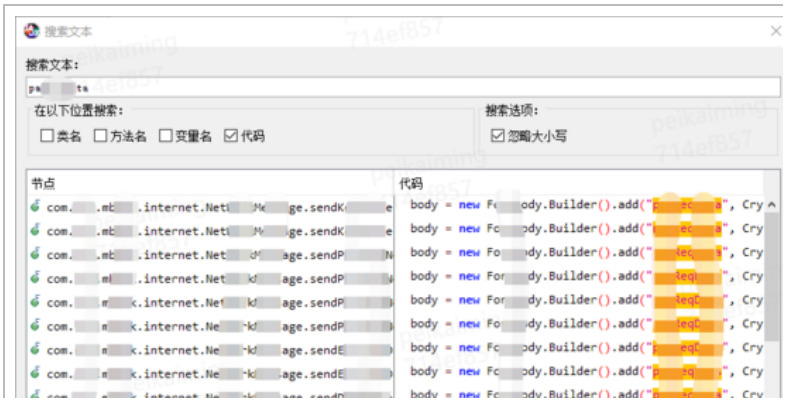
我们继续往下看，当打算测试转账、查询余额等功能时，发现数据包为加密状态，没有办法进行功能测试。

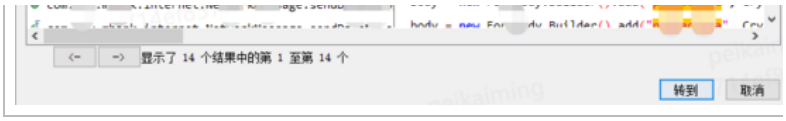
到了这里可能会难住一批人，但是，再硬的骨头也要试着啃一口，看我逆向大法破解加密算法，完成渗透测试。

首先我们可以使用 jadx、jeb、gda 等逆向分析工具，对 apk 文件进行反编译，这里以 jdax 为例：



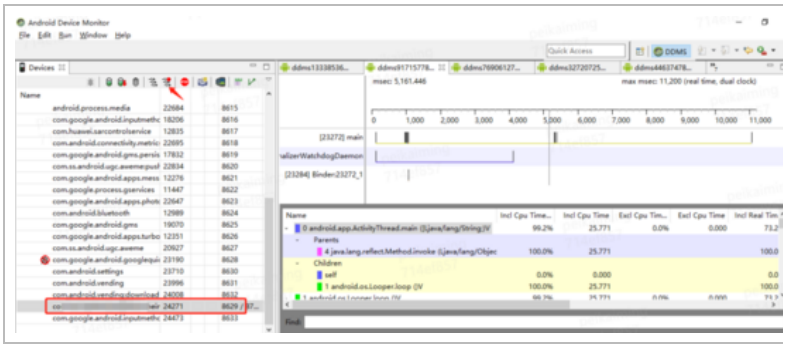
下面我们根据加密的数据包中的特殊字段，在代码中进行定位，很幸运的发现了如下 14 处，用到特殊字符串的位置，猜测是不同业务对应不同函数。



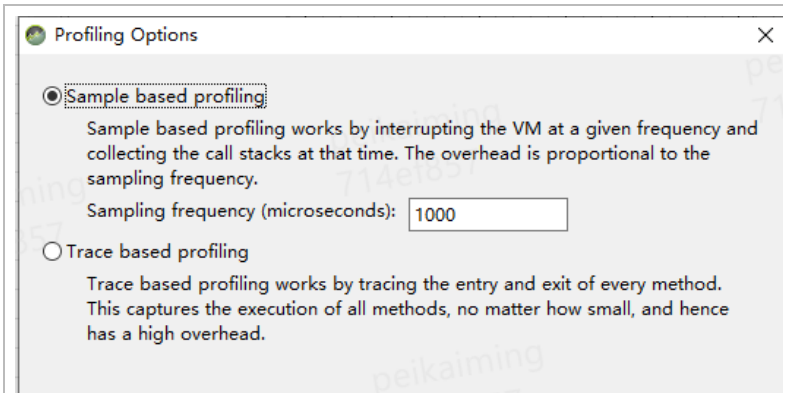


这其中肯定存在着我们所需要的函数，如果一个一个进行分析的话也能够发现最终调用的是哪个函数，但是为了方便，这里使用另一种方式进行定位，method profiling 工具 (位于 AndroidSDK 中，需要自行下载配置 SDK 环

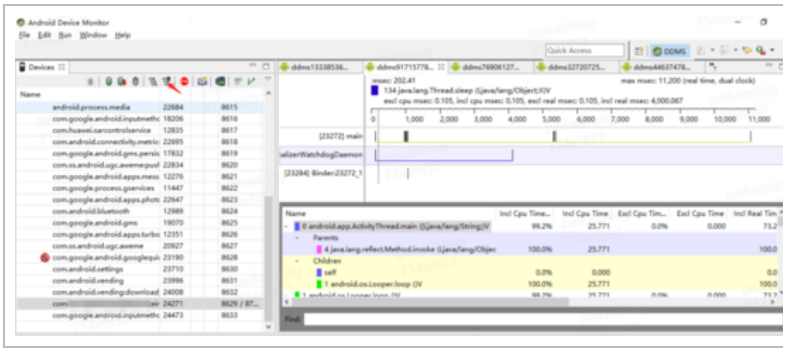
境)，下面实际操作下，首先选中我们要操作的 app 的进程，然后选择 start method profiling 后操作手机中的功能。



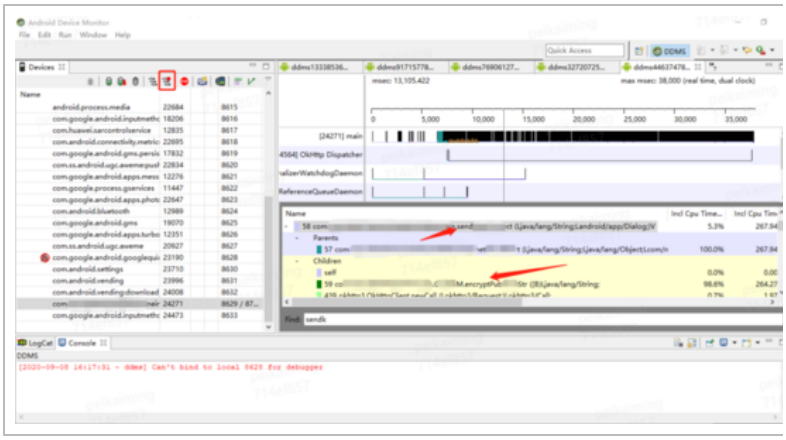
出现如下界面，选择 OK。



然后开始在手机上操作要测试的功能，完成后点击 stop method profiling 即可。



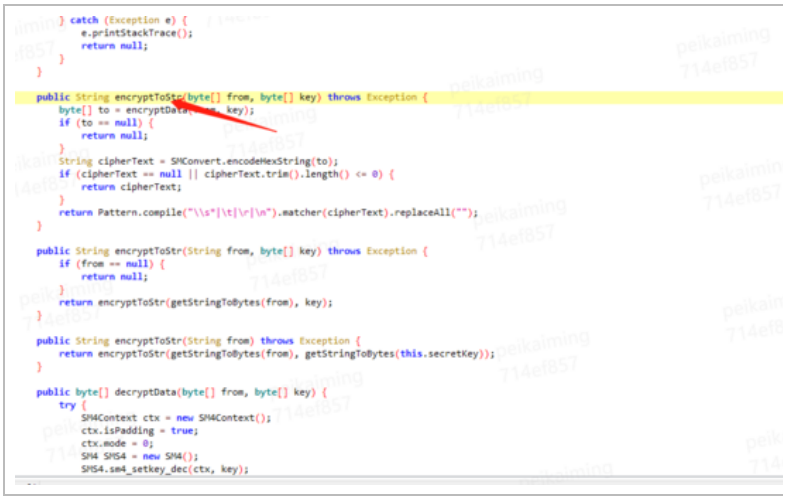
最终会在右侧展示出调用的所有函数，可以在其中对调用过的函数进行搜索，这里以上面从代码中定位到的函数进行搜索，结果如下：



在这里面可以看出来，函数内部还调用了一个 CrpytSM 类中的函数，对数据进行加密，然后才会调用 okhttp 对数据进行发送。下面我们回到反编译后的代码中，直接定位函数进行分析，根据调用栈发现，函数是走的如下代码。

我们跟进去继续分析，可以发现使用的是 SM2 算法对本次请求过程使用的密钥进行加密后，传输到服务端。

而在转账、查询等操作时都是使用的 SM2 加密传输的 key 和 SM4 进行加密传输，最终调用如下函数进行加密。



```
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    public String encryptToStr(byte[] from, byte[] key) throws Exception {
        byte[] to = encryptData(from, key);
        if (to == null) {
            return null;
        }
        String cipherText = SMConvert.encodeHexString(to);
        if (cipherText == null || cipherText.trim().length() <= 0) {
            return cipherText;
        }
        return Pattern.compile("\\s*|\t|\r|\n").matcher(cipherText).replaceAll("");
    }

    public String encryptToStr(String from, byte[] key) throws Exception {
        if (from == null) {
            return null;
        }
        return encryptToStr(getStringToBytes(from), key);
    }

    public String encryptToStr(String from) throws Exception {
        return encryptToStr(getStringToBytes(from), getStringToBytes(this.secretKey));
    }

    public byte[] decryptData(byte[] from, byte[] key) {
        try {
            SM4Context ctx = new SM4Context();
            ctx.isPadding = true;
            ctx.mode = 0;
            SM4 SMS4 = new SM4();
            SMS4.sm4_setkey_dec(ctx, key);
```

至此，我们梳理清楚了客户端加密的关键逻辑，其实对于加密，开发人员为了降低耦合度，每一个客户端都有统一的加密接口，也就是所有的入参都会经过这个加密接口，加密之后会再次输出，所以大家在梳理逻辑的时候一定要找准位置，单独分析用一个点（比如登录），摸透之后就会得到想要的结果。接下来，我们直接通过 hook 这个加密函数，就可以控制整个程序的所有入参及加密后的出参。

Frida Hook 大法

Frida 在之前的文章中有过详细的介绍，这儿就不详细讲他的用法了，如果不是很了解的可以看看之前发布的文章，或者在通过其他渠道学习学习。这里主要是基于前面的理论，在这里加入实战演练。下面开始使用 frida hook 并修改加密的内容。首先我们看下 key 是怎么生成的，经过分析后 hook 调用 createKey 函数就可以知道，app 这一次运行阶段使用的 key，具体 hook 代码如下：

```
if (Java.available) {
    console.log("********** hook start ***********");
    Java.perform( fn: function() {
        var CryptSM = Java.use( className: "co                    CryptSM");
        CryptSM.createKey.implementation=function(){
            console.log("********** createKey  start ***********");
            var result = this.createKey();
            var Arrays = Java.use( className: "java.util.Arrays");
            console.log("result is : " + Arrays.toString(result));
            console.log("********** createKey  end ***********")
            return result;
        }
```

返回值即这一次使用的 key，下面看下加密。

```
var strs = "未加密的参数信息";
var SM4EcbUtil = Java.use( className: "c                    SM4EcbUtil")
SM4EcbUtil.encryptToStr.overload( args: "[B","[B").implementation = function (from, key){
    console.log("************** SM4EcbUtil encryptToStr start **************")
    var from = this.getSt      Bytes(strs)
    var result =  this.encryptToStr(from, key)
    var String = Java.use( className: "java.lang.String")
    var array = Java.use( className: "java.util.Arrays")
    var data = String.$new(from)
    console.log("from data: " + data)
    data = String.$new(key)
    console.log("sm4 key: " + array.toString(key))
    console.log("encode result : " + result)
    console.log("************** SM4EcbUtil encryptToStr start **************")
    return result;
}
```

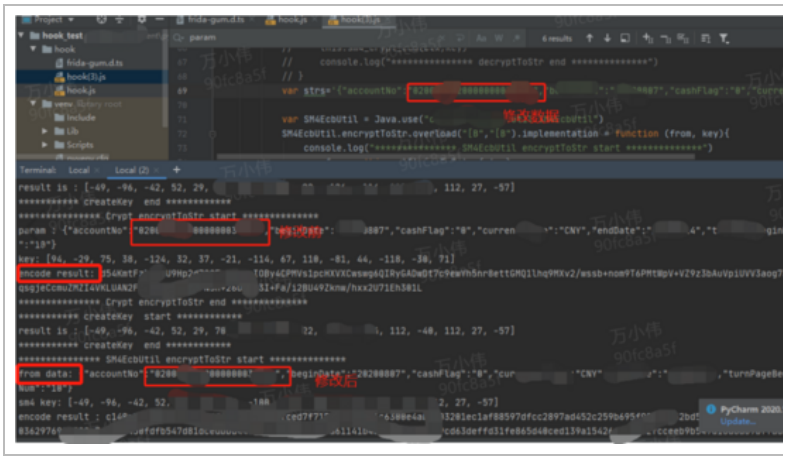
这里可以将 from 进行替换，实现参数修改，后面的越权转账、越权查询等都是基于此函数进行操作的，从这里可

以发现 打印的 sm4 key 和上面生成的 key 是相同的。

测试过程中，我们发现了该 APP 可以越权操作。水平越权查询，越权转账。越权在金融行业是一个非常严重的漏洞，直接影响着资金安全及客户信息安全。

下面我们用一个越权查询交易明细来验证我们的整个渗透思路的正确性。

启动 Frida，点击 APP 需要测试的功能，我们直接 hook 上文中梳理出的程序统一入参接口 xxx.xxx.SM4EcbUtil 加密算法。如下图：



上图中，Param 参数即加密前的参数，我们直接用 strs 替换该参数，将其中的账号修改为另外的一个账号，修改后为 from data 参数。其实经过分析，第一个加密过程得出的结果（encode result）也传到了后端，数据包响应 200，个人觉得是一个签名校验的过程。第二个 SM4 加密的数据包才是真正业务逻辑处理的过程。此处的架构设计没有将解密和验签放在同一个数据包，也具有一定的风险。

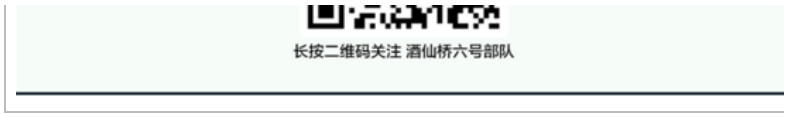
返回结果如下：


7:17
交易明细
账 号：
起始日期：
结束日期：
查询
转账存入
2022-02-08
实际为 1明细

总结

客户端安全其实是作用在客户端每一个环节，如加壳、加密、签名、证书认证等，每一个点也可能是形成木桶原理的那个点。渗透过程中，应该层层分析，不放过任何可能出现问题的点。一旦一个点沦陷，即可造成其他点的沦陷。各位师傅在测试过程中遇到无法抓包、数据包加密、

程序加壳、签名校验等问题时，不要慌，可以先搜一波，相信大部分问题都会解决，关于客户端的渗透，逆向大法是必不可少的，只要细心都能够分析出来的。如果还不会逆向、frida 的那就得抓紧学习了，要跟上时代得步伐。



全文完