原文地址:http://drops.wooyun.org/tips/2871

0×00 起因

最近想了解下移动端app是如何与服务端进行交互的，就顺手下了一个某app抓下http包。谁知抓下来的http居然都长这个模样：

```
POST /ca?qrt=***LoginHTTP/1.1
Content-Type:application/x-www-form-urlencoded
Content-Length:821
Host:client.XXX.com
Connection:Keep-Alive

c=B946D7CF7B9E5B589F8DE6BC9E5DACF08DA6B35DA7A899DCA5AD5A58ADDAD5E66363589EF2D385B1ACACABDAD5E65F63589EF2D3F5B43EA7ACE36DFCA5383EABE7D4F1403E379FF0DEFCAD9FABA7E6E1F243A
```

这是我在尝试登陆时抓包获取的唯一http包，显而易见POST数据中的c参数是包含登录信息的，但是为什么长这个模样？为了得到答案，我开启了我一周的Android动态调试和静态分析学习之旅。

这篇文章将通过这段字符串原文的过程，向各位介绍几种非常好用的Android调试工具以及它们的一些简单用法。

0×01 分析过程

1.基本静态分析过程

拿到一个apk最常规的做法应该是就是，反编译查看一下java源码了。

用apktool反编译得到smali（其实主要是为了看AndroidManifest.xml）：

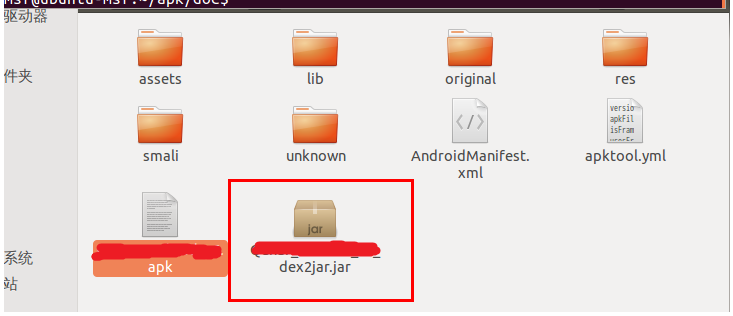
```
/*我用的apktool是2.0.0-Beta9，命令和常见的1.x版本的命令有所不同*/
apktool d XXX.apk -o ./doc
/*我用的apktool是2.0.0-Beta9，命令和常见的1.x版本的命令有所不同*/
apktooldXXX.apk-o./doc
```

```
msf@ubuntu-msf:~/apk$ apktool d [redacted] -o ./doc
I: Using Apktool 2.0.0-Beta9 on [redacted]
I: Loading resource table...
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/msf/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
Cleaning up unclosed ZipFile for archive /home/msf/apktool/framework/1.apk
I: Decoding values */* XMLs...
I: Baksmaling...
I: Copying assets and libs...
I: Copying unknown files/dir...
I: Copying original files...
```

然后用的dex2jar工具将apk反编译为jar，并通过JD-GUI来查看java源码：

```
dex2jar.shXXX.apk
```

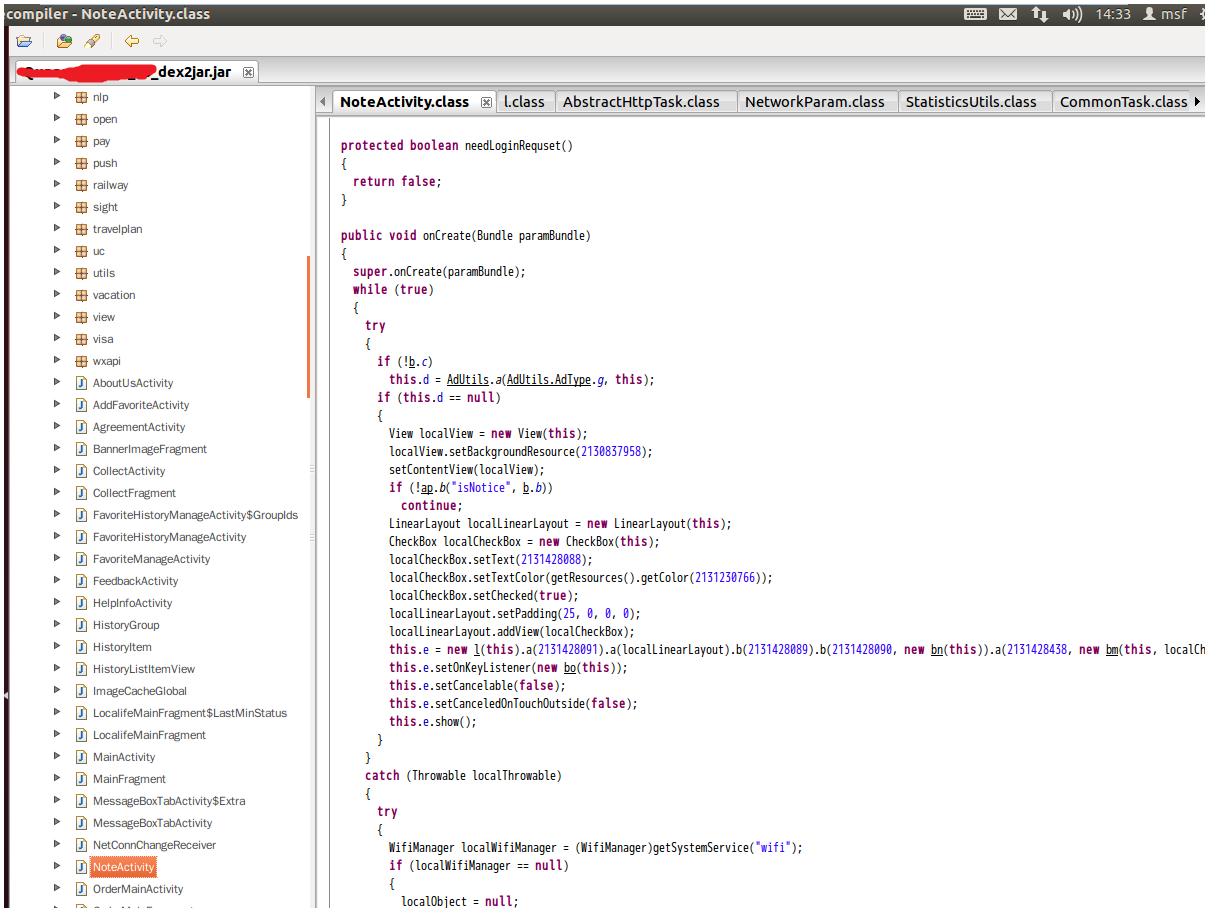
```
msf@ubuntu-msf:~/apk/doc$ dex2jar.sh [redacted].apk
this cmd is deprecated, use the d2j-dex2jar if possible
dex2jar version: translator-0.0.9.15
dex2jar [redacted].apk -> [redacted]_dex2jar.jar
Done.
msf@ubuntu-msf:~/apk/doc$
```



在apktool反编译的目录中我们可以翻看AndroidManifest.xml来了解apk文件的基本结构，我从中首先找到主Activity的所在：

```
<activityandroid:configChanges="keyboardHidden|orientation"android:exported="true"android:name="com.XXX.NoteActivity"android:screenOrientation="portrait"android:theme=
```

可以从上面的内容看出主Activity是com.XXX.NoteActivity这个类所定义的，然后通过JD-GUI打开dex2jar反编译后的jar包，查看NoteActivity。



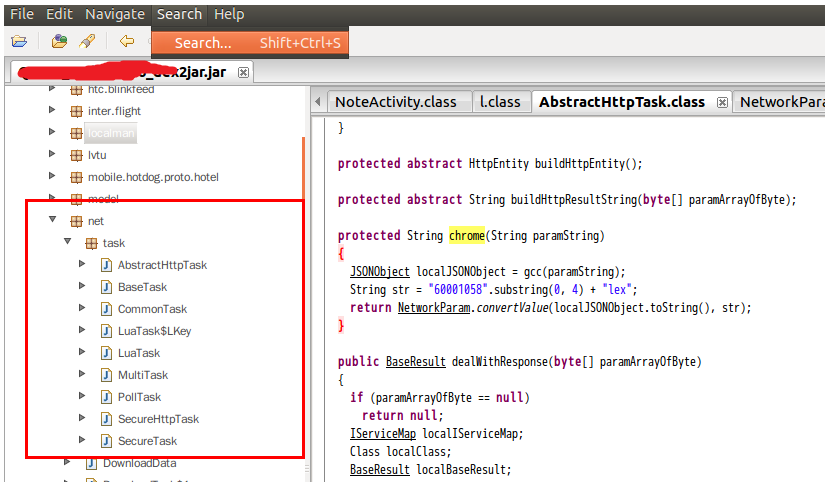
先来看onCreate中的操作，发现使用produard对apk进行了混淆了。这种混淆虽然不会影响我们反编译出来的代码内容，但是由于对类名、函数名、变量名进行了随机命名，导致我们阅读代码的过程比较痛苦。

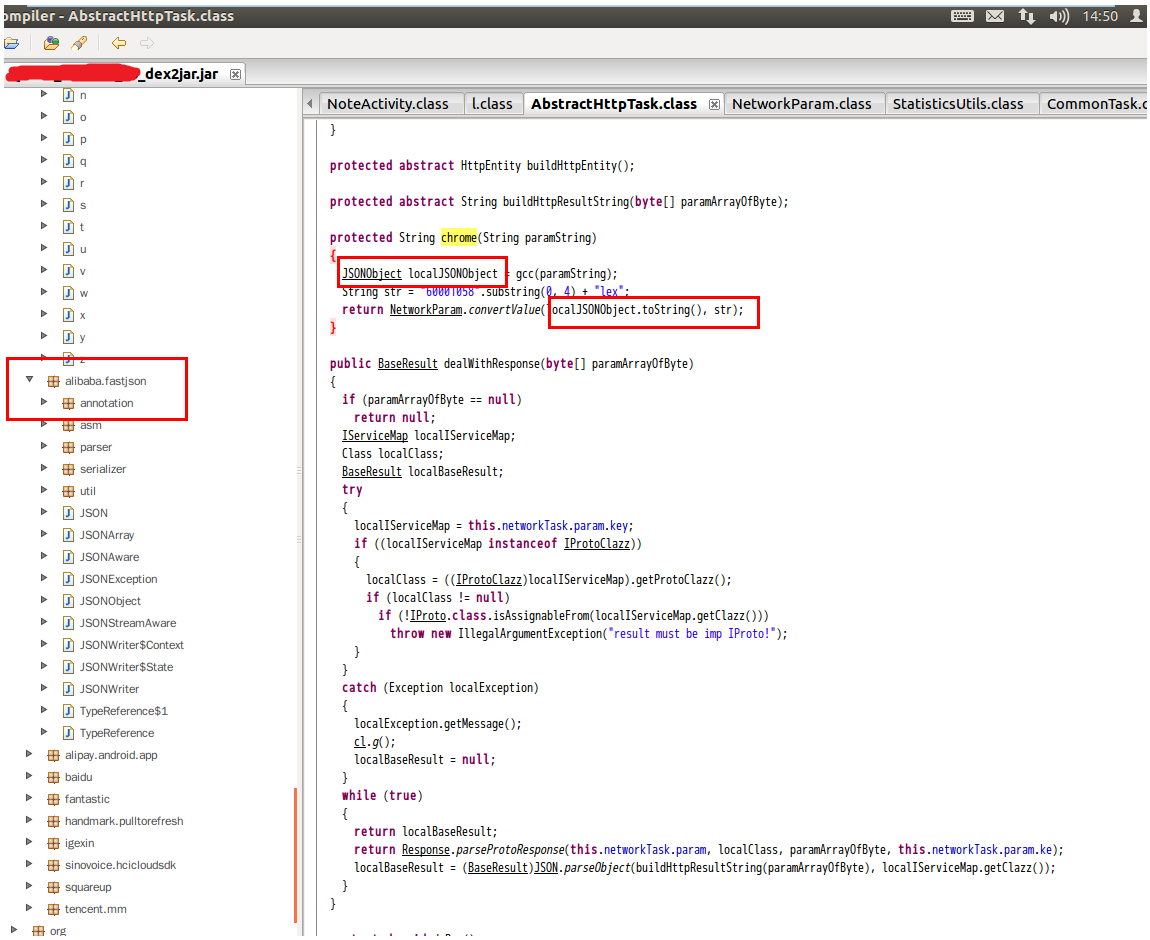
我的目的很明确，就是定位到处理提交数据的代码，没有必要去花大量的时间来阅读被混淆的代码，所以我决定使用动态跟踪程序运行的轨迹来定位我想要获得的代码。

2.动态定位过程

虽然要用动态的方法来定位，但是还是需要简单的阅读java源码来确定提交数据的大概处理方式。

我的运气还是不错，网络传输部分的代码并没有被混淆。大体看了一下这些代码，发现和一般的app一样，客户端和服务端的数据交互也是使用json的格式进行的，并且使用了阿里开源的fastjson类来处理json内容。



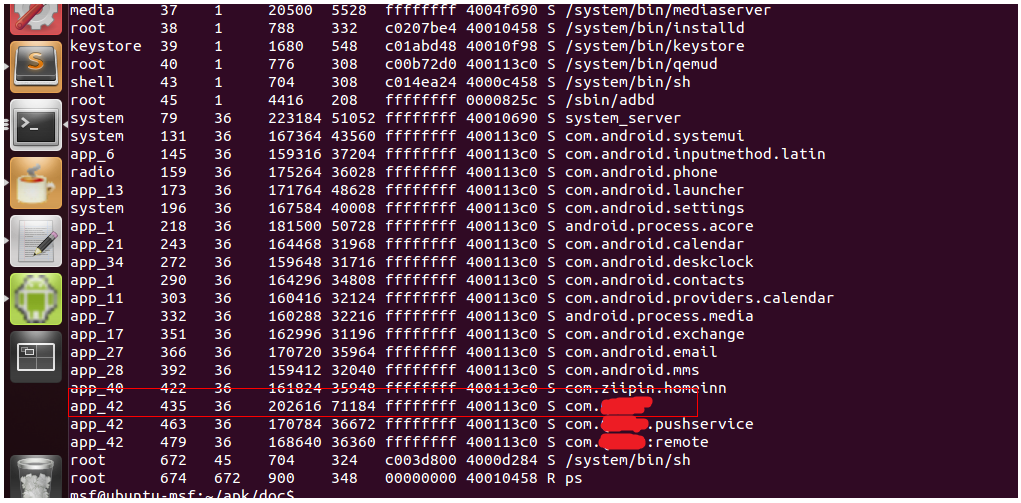


了解了以上的这些情况，我决定通过跟踪JSONObject这个类来定位处理提交数据的位置。

这里推荐一个分析app的神器——Andbug，虽然不能用来单步调试，但是动态跟踪app中各种线程调用栈、类调用栈、方法调用栈，断点获取当前内存中变量内容等功能还是非常实用的。

废话不多说了，来看操作吧，先获取要动态分析的app进程ID：

```
adb shell ps
```



进程ID 445，使用Andbug挂载该进程，并使用classes命令查找fastjson类的全路径：

andbug shell -p 435 classes JSONObject andbug shell -p435 classes JSONObject

```
## AndBug (C) 2011 Scott W. Dunlop <swdunlop@gmail.com>
>> classes JSONObject
## Loaded Classes
   -- com.alibaba.fastjson.parser.deserializer.JSONObjectDeserializer
   -- org.json.JSONObject
   -- com.alibaba.fastjson.JSONObject
>>
```

这里提示一个使用classes和method命令查找的小技巧。我们在Andbug的shell环境下使用classes时很容易由于class过多而导致没办法看到所有的class。这是我们可以在终端环境下使用classes命令配合more来一点点的查看，就像这样： andbug classes -p 435|more

然后我们使用class-trace命令来对这个类进行跟踪：

```
class-trace com.alibaba.fastjson.JSONObject
```

```
>> class-trace com.alibaba.fastjson.JSONObject
## Setting Hooks
   -- Hooked com.alibaba.fastjson.JSONObject
>>
```

在app中随便触发一个会提交请求的事件，调用过程在终端中完美的呈现了出来：

```
## trace thread <13> AsyncTask #3       (running suspended)
   -- com.alibaba.fastjson.JSONObject.put(Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/Object;:0
      -- this=Lcom/alibaba/fastjson/JSONObject; <830036424072>
   -- com.[redacted].net.task.AbstractHttpTask.gcc(Ljava/lang/String;)Lcom/alibaba/fastjson/JSONObject;:146
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.AbstractHttpTask.chrome(Ljava/lang/String;)Ljava/lang/String;:0
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.CommonTask.buildHttpEntity()Lorg/apache/http/HttpEntity;:73
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.AbstractHttpTask.getResult()Lcom/[redacted]/model/response/BaseResult;:18
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.BaseTask.doInBackground([Ljava/lang/Void;)Lcom/[redacted]/model/response/BaseResult;:92
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.BaseTask.doInBackground([Ljava/lang/Object;)Ljava/lang/Object;:2
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].utils.c.call()Ljava/lang/Object;:21
      -- this=Lcom/[redacted]/utils/c; <830039388176>
   -- java.util.concurrent.FutureTask$Sync.innerRun()V:23
      -- this=Ljava/util/concurrent/FutureTask$Sync; <830036564256>
   -- java.util.concurrent.FutureTask.run()V:2
      -- this=Lcom/Qunar/utils/d; <830039388200>
   -- java.util.concurrent.ThreadPoolExecutor.runWorker(Ljava/util/concurrent/ThreadPoolExecutor$Worker;)V:28
      -- this=Ljava/util/concurrent/ThreadPoolExecutor; <830028345088>
      -- completedAbruptly=True
      -- task=Lcom/Qunar/utils/d; <830039388200>
      -- w=Ljava/util/concurrent/ThreadPoolExecutor$Worker; <830024737928>
   -- java.util.concurrent.ThreadPoolExecutor$Worker.run()V:2
      -- this=Ljava/util/concurrent/ThreadPoolExecutor$Worker; <830024737928>
   -- java.lang.Thread.run()V:6
      -- this=Ljava/lang/Thread; <830022442624>
## trace thread <13> AsyncTask #3       (running suspended)
   -- com.alibaba.fastjson.JSONObject.put(Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/Object;:0
      -- this=Lcom/alibaba/fastjson/JSONObject; <830036424072>
   -- com.[redacted].net.task.AbstractHttpTask.gcc(Ljava/lang/String;)Lcom/alibaba/fastjson/JSONObject;:162
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.AbstractHttpTask.chrome(Ljava/lang/String;)Ljava/lang/String;:0
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.CommonTask.buildHttpEntity()Lorg/apache/http/HttpEntity;:73
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.AbstractHttpTask.getResult()Lcom/[redacted]/model/response/BaseResult;:18
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.BaseTask.doInBackground([Ljava/lang/Void;)Lcom/[redacted]/model/response/BaseResult;:92
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].net.task.BaseTask.doInBackground([Ljava/lang/Object;)Ljava/lang/Object;:2
      -- this=Lcom/[redacted]/net/task/CommonTask; <830039388112>
   -- com.[redacted].utils.c.call()Ljava/lang/Object;:21
      -- this=Lcom/[redacted]/utils/c; <830039388176>
   -- java.util.concurrent.FutureTask$Sync.innerRun()V:23
      -- this=Ljava/util/concurrent/FutureTask$Sync; <830036564256>
   -- java.util.concurrent.FutureTask.run()V:2
      -- this=Lcom/Qunar/utils/d; <830039388200>
   -- java.util.concurrent.ThreadPoolExecutor.runWorker(Ljava/util/concurrent/ThreadPoolExecutor$Worker;)V:28
```

可以看到调用过程都用到了com.XXX.net.task.CommonTask中的方法，打开这个类的java源码第一眼就看到这段代码：

```
#!java
protectedHttpEntity buildHttpEntity()
  {
    if(this.hostUrl.indexOf("?")>0)
      this.hostUrl=(this.hostUrl+"&qrt="+this.networkTask.param.key.getDesc());
    while(true)
    {
      Stringstr=String.valueOf(this.networkTask.param.ke);
      StringBuilderlocalStringBuilder1=newStringBuilder();
      localStringBuilder1.append("c="+chrome(str));
      localStringBuilder1.append("&");
      StringBuilderlocalStringBuilder2=newStringBuilder("b=");
      BaseParamlocalBaseParam=this.networkTask.param.param;
      SerializerFeature[]arrayOfSerializerFeature=newSerializerFeature[1];
      arrayOfSerializerFeature[0]=SerializerFeature.WriteTabAsSpecial;
      localStringBuilder1.append(NetworkParam.convertValue(JSON.toJSONString(localBaseParam,arrayOfSerializerFeature),str));
      if((this.networkTask.param.param instanceofHotelBookParam))
      {
        HotelBookParamlocalHotelBookParam=(HotelBookParam)this.networkTask.param.param;
        if(localHotelBookParam.vouchParam!=null)
          dealVouchRequest(localStringBuilder1,localHotelBookParam.vouchParam);
      }
      localStringBuilder1.append("&");
      localStringBuilder1.append("ext="+NetworkParam.convertValue(XXXApp.getContext().ext,str));
      localStringBuilder1.append("&v=alex");
      this.networkTask.param.url=localStringBuilder1.toString();
      try
      {
        StringEntitylocalStringEntity=newStringEntity(this.networkTask.param.url);
        returnlocalStringEntity;
        this.hostUrl=(this.hostUrl+"?qrt="+this.networkTask.param.key.getDesc());
      }
      catch(UnsupportedEncodingExceptionlocalUnsupportedEncodingException)
      {
        cl.m();
```

```
      }
    }
    returnnull;
  }
```

结合之前的抓包，这应该就是我要找的地方了。从中找到处理c参数的代码，看到调用了com.XXX.net.task.AbstractHttpTask.chrome对参数值进行了处理，跟进chrome方法：

```
#!java
protectedStringchrome(StringparamString)
  {
    JSONObjectlocalJSONObject=gcc(paramString);
    Stringstr="60001058".substring(0,4)+"lex";
    returnNetworkParam.convertValue(localJSONObject.toString(),str);
  }
```

继续赶进到convertValue方法：

```
#!java
publicstaticStringconvertValue(StringparamString1,StringparamString2)
  {
    if(TextUtils.isEmpty(paramString1))
      return "";
    if(paramString2==null)
      paramString2="";
    try
    {
      Stringstr=URLEncoder.encode(Goblin.e(paramString1,paramString2),"utf-8");
      returnstr;
    }
    catch(ThrowablelocalThrowable)
    {
      localThrowable.printStackTrace();
    }
    return "";
  }
```

感觉的胜利的曙光越来越近了，这个Goblin.e应该就是最后的加密方法了吧，谁知打开这个文件（内心一万只草泥马在狂奔）：

```
#!java
packageXXX.lego.utils;
importcom.XXX.XXXApp;

publicclassGoblin
{
  static
  {
    try
    {
      System.loadLibrary("goblin_2_5");
      return;
    }
    catch(UnsatisfiedLinkErrorlocalUnsatisfiedLinkError1)
    {
      try
      {
        System.load("/data/data/"+XXXApp.getContext().getPackageName()+"/lib/lib"+"goblin_2_5"+".so");
        return;
      }
      catch(UnsatisfiedLinkErrorlocalUnsatisfiedLinkError2)
      {
      }
    }
  }

  publicstaticnativeStringSHR();

  publicstaticnativeStringd(StringparamString1,StringparamString2);

  publicstaticnativeStringdPoll(StringparamString);

  publicstaticnativeStringda(StringparamString);

  publicstaticnativeStringdn(byte[]paramArrayOfByte,StringparamString);

  publicstaticnativebyte[]dn1(byte[]paramArrayOfByte,StringparamString);

  publicstaticnativeStringduch(StringparamString);

  publicstaticnativeStringe(StringparamString1,StringparamString2);

  publicstaticnativeStringePoll(StringparamString);

  publicstaticnativeStringea(StringparamString);

  publicstaticnativebyte[]eg(byte[]paramArrayOfByte);

  publicstaticnativeStringes(StringparamString);

  publicstaticnativeintgetCrc32(StringparamString);

  publicstaticnativeStringgetPayKey();

  publicstaticnativeStringve(StringparamString);
}
```

3.调用so文件函数

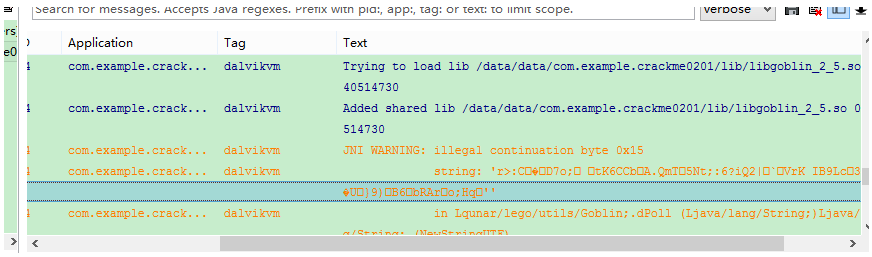
居然把加密方法写到了so文件中！难道要去看ARM汇编？

既然这个so文件中有加密函数，那是不是就应该有解密函数，那我应该还是可以偷懒的吧。

我们在上面看到的e函数肯定是用来加密的，那那个d函数是不是用来解密的（encode和decode）？

自己本地创建一个app，并且创建一个XXX.lego.utils包，添加一个Goblin.java文件，把我们刚刚看到的Goblin源码粘贴进去。然后在app的一个Activity中导入Goblin，并在OnCreate中调用d函数来尝试解密。部分代码如下：

```
#! java
Stringc="B946D7CF7B9E5B589F8DE6BC9E5DACF08DA6B35DA7A899DCA5AD5A58ADDAD5E66363589EF2D385B1ACACABDAD5E65F63589EF2D3F5B43EA7ACE36DFCA5383EABE7D4F1403E379FF0DEFCAD9FABA7E6
Stringp2="6000lex";
Stringtest=Goblin.d(c,p2);
System.out.println(test);
```



天不遂人愿啊！解密出错，看来真的要去看ARM汇编了。。。。。。

4.动态调试so文件

由于app自带的加密数据，我们不知道原来的样子，所以要自己构造一个字符串加密，来调试。修改上面的app代码如下：

```
#! java
Stringjson="json{/"test/":/"test1/",/"test4/":/"test1/",/"test5/":/"test1/",/"test6/":/"test1/",/"test3/":/"test1/",/"test2/":/"test1/",/"test1/":/"test1/"}";
Stringp2="6000lex";
Stringtest=Goblin.e(c,p2);
System.out.println(test);
```

生成代码如下：

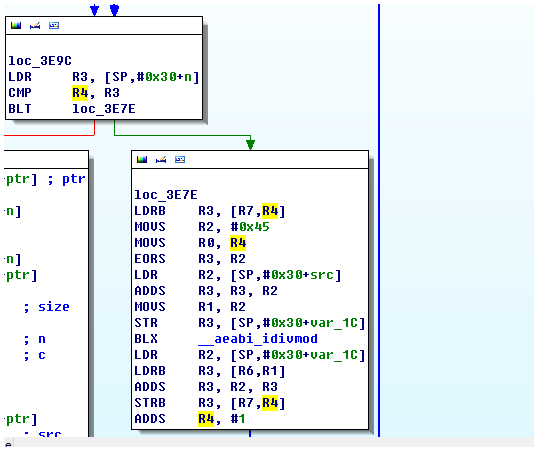
```
C0990850B69C969E92C19C9DC5CDD9F0FAB99AD3960CE3F6D2CFB0AA9AE904F6C0A099A68DFFD0C1EE978CA985CAD2FCF6CD92A7C4FCE106CCC99CC39C11F7F3D0A9BDAD8AE3E0D9C3BC898EB8DCD3F2BB8B8BB
```

好了准备活动完成了，下面我们开始动态跟踪之旅吧。在《Android软件安全与逆向分析》中提供的动态分析工具是IDA pro 6.1以上版本，这个我在调试过程中发现加载很慢。虽然加载完成后，能够跟着IDA生成的流程图来调试很爽，但是加载成功率实在是太低了。所以，我放弃了用IDA进行动态调试，而是选择了 这个号称移动端Onllydbg的gikdbg来进行调试，同时配合IDA的流程图。

gikdbg使用参考《gikdbg.art系列教程2.1-调试so动态库》这篇blog很容易上手，这里也就不多说了。

调试跟踪过程很枯燥，也没什么可以说的，我们直接看结过吧。

通过反复的动态跟踪，确定下面这个循环是加密的关键：



可以看出加密方法比较简单，对于源数据的每一个字符与0×45进行异或，然后jia0x24，最后再加上硬编码在so文件的一串key中的一个字符。根据汇编逆向出来的python代码如下：

```
result=""
i=0
while(i<len(json)):
    char=json[i]^69
    j=i
    ifj>len(key):
        j=j%len(key)
    encode=int(ord(char))+36+key[i]
    result+=encode
    i+=1
```

在加密完数据后，会在数据头部添加一个8个字符（32位）的校验数据，校验算法使用的是adler32。由此可以推出解密算法，代码如下：

defdecode():

```
ejson='B69C969E92C19C9DC5CDD9F0FAB99AD3960CE3F6D2CFB0AA9AE904F6C0A099A68DFFD0C1EE978CA985CAD2FCF6CD92A7C4FCE106CCC99CC39C11F7F3D0A9BDAD8AE3E0D9C3BC898EB8DCD3F2BB8B8BB3

key='cBHO06GYkxNModVyAtXiGzlPETyS5KUL8gE4'
i=0
result=''
while(i<len(ejson)):
    j=i/2
    char=ejson[i:i+2]
    ifj<len(key):
        k=key[j]
    else:
        k=key[j%len(key)]
    i=i+2
    c=int(char,16)-int(ord(k))
    ifc<0:
        c+=128
    c=c-36
    ifc<0:
        c+=128
    c=c^69
    dchar=chr(c)
    result+=dchar
```

```
printresult
decode()
```

其中ejson中的内容为，我们使用e函数加密后获得的内容剔除前八位，解密效果如下：

0×02 最终结果&分析总结

不过悲剧的是用这个解密方法没办法解密前面我抓包获取的数据。。。。。。

郁闷之心无以言表啊！！！

不过这个过程还是很有意义的，了解了Android各种姿势的动态调试方法。这里再次回顾一些这个过程。

首先通过反编译获取smali和java代码进行静态分析，发现代码被混淆后，明确自己的最终目标——找到处理提交请求的方法，然后进行动态跟踪。动态跟踪和静态分析结合定位出处理提交请求的几个类，翻看这些类的代码，来找到最终我们想找的方法。

在发现处理方法使用了so文件中的函数，通过自己构造app来分别调用so中的各个函数，试图从中找到直接的解密函数。

在so中没有找到解密函数的情况下，通过动态调试与静态查看汇编，分析出加密算法，并写出解密工具。

0×03 参考文章

【1】《Assembly Programming Principles》【2】《Android动态逆向分析工具（一）——Andbug之基本操作》【3】《Android动态逆向分析工具（四）—— Andbug补充调试功能》
【4】《gikdbg.art系列教程2.1-调试so动态库》