



ANDROID APPLICATIONS REVERSING 101

2017-04-27 |  #Dex2Jar, #Frida, #Hopper, #IDA, #JADX, #JD-GUI, #Xposed, #Xposed Framework, #android, #apk, #apktool, #dalvik, #dex, #re, #reversing



Every day we see a bunch of new Android applications being published on the Google Play Store, from games, to utilities, to IoT devices clients and so forth, almost every single aspect of our life can be somehow controlled with “an app”. We have smart houses, smart fitness devices and smart coffee machines ... but is this stuff just smart or is it **secure** as well? :)

Reversing an Android application can be a (relatively) easy and fun way to answer this question, that’s why I decided to write this blog post where I’ll try to explain the basics and give you some of my “tricks” to reverse this stuff faster and more effectively.

I’m not going to go very deep into technical details, you can learn yourself how Android works, how the Dalvik VM works and so forth, this is gonna be a **very basic practical guide** instead of a post full of theoretical stuff but no really useful contents.

Let’s start! :)



Prerequisites

In order to follow this introduction to APK reversing there're a few prerequisites:

- A working brain (I don't give this for granted anymore ...).
- An Android smartphone (doh!).
- You have a basic knowledge of the Java programming language (you understand it if you read it).
- You have the JRE installed on your computer.
- You have adb installed.
- You have the Developer Options and USB Debugging enabled on your smartphone.

What is an APK?

An Android application is packaged as an **APK** (*Android Package*) file, which is essentially a ZIP file containing the compiled code, the resources, signature, manifest and every other file the software needs in order to run.

Being it a ZIP file, we can start looking at its contents using the `unzip` command line utility (or any other unarchiver you use):



```
unzip application.apk -d application
```

Here's what you will find inside an APK.

```
/AndroidManifest.xml (file)
```

This is the binary representation of the XML manifest file describing what permissions the application will request (keep in mind that some of the permissions might be requested at runtime by the app and not declared here), what activities (GUIs) are in there, what services (stuff running in the background with no UI) and what receivers (classes that can receive and handle system events such as the device boot or an incoming SMS).

Once decompiled (more on this later), it'll look like this:

```
1  <?xml version="1.0" encoding="utf-8" standalone="no"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.company.appname"
3                                     platformBuildVersionCode="24"
4                                     platformBuildVersionName="7.0">
5      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
6      <uses-permission android:name="android.permission.INTERNET"/>
7
8      <application android:allowBackup="true" android:icon="@mipmap/ic_launcher"
9                  android:label="@string/app_name"
10                 android:supportsRtl="true" android:theme="@style/AppTheme">
11          <activity android:name="com.company.appname.MainActivity">
12              <intent-filter>
13                  <action android:name="android.intent.action.MAIN"/>
```

```
14         <category android:name="android.intent.category.LAUNCHER"/>
15     </intent-filter>
16 </activity>
17 </application>
18
19 </manifest>
```

Keep in mind that this is the perfect starting point to isolate the application “entry points”, namely the classes you’ll reverse first in order to understand the logic of the whole software. In this case for instance, we would start inspecting the `com.company.appname.MainActivity` class being it declared as the main UI for the application.

```
/assets/* ( folder )
```

This folder will contain application specific files, like wav files the app might need to play, custom fonts and so on. Reversing-wise it’s usually not very important, unless of course you find inside the software functional references to such files.

```
/res/* ( folder )
```

All the resources, like the activities xml files, images and custom styles are stored here.

```
/resources.arsc ( file )
```

This is the “index” of all the resources, long story short, at each resource file is assigned a numeric identifier that the app will use in order to identify that specific entry and the `resources.arsc` file maps these files to their identifiers ... nothing very interesting about it.

```
/classes.dex ( file )
```

This file contains the Dalvik (the virtual machine running Android applications) bytecode of the app, let me explain it better. An Android application is (most of the times) developed using the Java programming language. The java source files are then compiled into this bytecode which the Dalvik VM eventually will execute ... pretty much what happens to normal Java programs when they're compiled to `.class` files.

Long story short, this file contains the logic, that's what we're interested into.

Sometimes you'll also find a `classes2.dex` file, this is due to the DEX format which has a limit to the number of classes you can declare inside a single dex file, at some point in history Android apps became bigger and bigger and so Google had to adapt this format, supporting a secondary `.dex` file where other classes can be declared.

From our perspective it doesn't matter, the tools we're going to use are able to detect it and append it to the decompilation pipeline.

```
/libs/ ( folder )
```

Sometimes an app needs to execute native code, it can be an image processing library, a game engine or whatever. In such case, those `.so` ELF libraries will be found inside the `libs` folder, divided into architecture specific subfolders (so the app will run on ARM, ARM64, x86, etc).

```
/META-INF/ ( folder )
```

Every Android application needs to be signed with a developer certificate in order to run on a device, even debug builds are signed by a debug certificate, the `META-INF` folder contains information about the files inside the APK and about the developer.

Inside this folder, you'll usually find:

- A `MANIFEST.MF` file with the SHA-1 or SHA-256 hashes of **all** the files inside the APK.
- A `CERT.SF` file, pretty much like the MANIFEST.MF, but signed with the `RSA` key.
- A `CERT.RSA` file which contains the developer public key used to sign the `CERT.SF` file and digests.

Those files are very important in order to guarantee the APK integrity and the ownership of the code. Sometimes inspecting such signature can be very handy to determine who really developed a given APK. If you want to get information about the developer, you can use the `openssl` command line utility:

```
openssl pkcs7 -in /path/to/extracted/apk/META-INF/CERT.RSA -inform DER -print
```

This will print an output like:

```
PKCS7:
  type: pkcs7-signedData (1.2.840.113549.1.7.2)
  d.sign:
    version: 1
    md_algs:
      algorithm: sha1 (1.3.14.3.2.26)
      parameter: NULL
    contents:
      type: pkcs7-data (1.2.840.113549.1.7.1)
      d.data: <ABSENT>
    cert:
      cert_info:
        version: 2
        serialNumber: 10394279457707717180
        signature:
          algorithm: sha1WithRSAEncryption (1.2.840.113549.1.1.5)
          parameter: NULL
        issuer: C=TW, ST=Taiwan, L=Taipei, O=ASUS, OU=PMD, CN=ASUS AMAX Key/emailAddress=admin@asus.com
        validity:
          notBefore: Jul  8 11:39:39 2013 GMT
          notAfter: Nov 23 11:39:39 2040 GMT
        subject: C=TW, ST=Taiwan, L=Taipei, O=ASUS, OU=PMD, CN=ASUS AMAX Key/emailAddress=admin@asus.com
```



```
key:
  algor:
    algorithm: rsaEncryption (1.2.840.113549.1.1.1)
    parameter: NULL
  public_key: (0 unused bits)
  ...
  ...
  ...
```

This can be gold for us, for instance we could use this information to determine if an app was really signed by (let's say) Google or if it was resigned, therefore modified, by a third party.

How do I get the APK of an app?

Now that we have a basic idea of what we're supposed to find inside an APK, we need a way to actually get the APK file of the application we're interested into. There are two ways, either you install it on your device and use `adb` to get it, or you use an online service to download it.

Pulling an app with ADB

First of all let's plug our smartphone to the USB port of our computer and get a list of the installed packages and their namespaces:

```
adb shell pm list packages
```

This will list all packages on your smartphone, once you've found the namespace of the package you want to reverse (`com.android.systemui` in this example), let's see what its physical path is:

```
adb shell pm path com.android.systemui
```

Finally, we have the APK path:

```
package:/system/priv-app/SystemUIGoogle/SystemUIGoogle.apk
```



Let's pull it from the device:

```
adb pull /system/priv-app/SystemUIGoogle/SystemUIGoogle.apk
```

And here you go, you have the APK you want to reverse!

Using an Online Service

Multiple online services are available if you don't want to install the app on your device (for instance, if you're reversing a malware, you want to start having the file first, then installing on a clean device only afterwards), here's a list of the ones I use:

- [Apk-DL](#)
- [Evozi Downloader](#)
- [Apk Leecher](#)

Keep in mind that once you download the APK from these services, it's a good idea to check the developer certificate as previously shown in order to be 100% sure you downloaded the correct APK and not some repackaged and resigned stuff full of ads and possibly malware.

Network Analysis

Now we start with some tests in order to understand what the app is doing while executed. My first test usually consists in inspecting the network traffic being generated by the application itself and, in order to do that, my tool of choice is [bettercap](#) ... well, that's why I developed it in the first place :P

Make sure you have bettercap installed and that both your computer and the Android device are on the same wifi network, then you can start MITM-ing the smartphone (`192.168.1.5` in this example) and see its traffic in realtime from the terminal:


```
sudo bettercap -T 192.168.1.5 -X
```

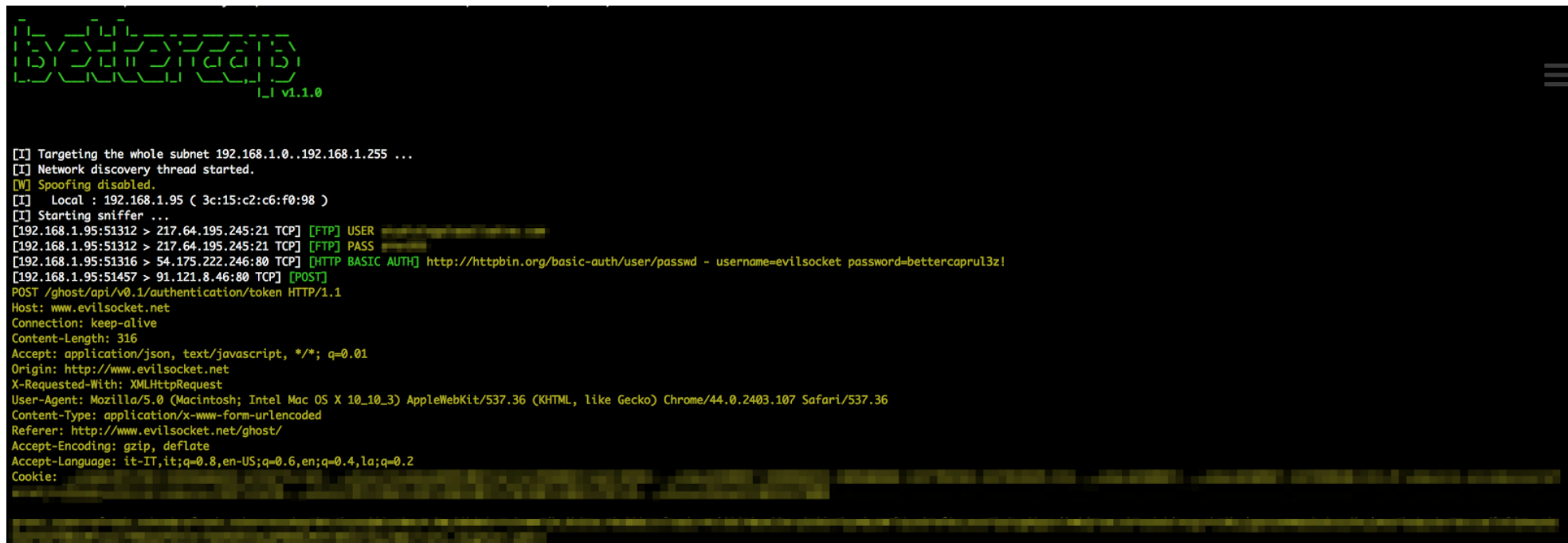
The `-X` option will enable the sniffer, as soon as you start the app you should see a bunch of HTTP and/or HTTPS servers being contacted, now you know who the app is sending the data to, let's now see **what** data it is sending:

```
sudo bettercap -T 192.168.1.5 --proxy --proxy-https --no-sslstrip
```

This will switch from passive sniffing mode, to proxying mode. All the HTTP and HTTPS traffic will be intercepted (and, if needed, modified) by bettercap.

If the app is correctly using public key pinning (as **every application should**) you will **not** be able to see its HTTPS traffic but, unfortunately, in my experience this only happens for a very small number of apps.

From now on, keep triggering actions on the app while inspecting the traffic (you can also use `Wireshark` in parallel to get a `PCAP` capture file to inspect it later) and after a while you should have a more or less complete idea of what protocol it's using and for what purpose.



Static Analysis

There're different tools you can use for this purpose, let's take a look at the most popular ones.

APKTool is the very first tool you want to use, it is capable of decompiling the `AndroidManifest` file to its original XML format, the `resources.arsc` file and it will also convert the `classes.dex` (and `classes2.dex` if present) file to an intermediary language called `SMALI`, an ASM-like language used to represent the Dalvik VM opcodes as a human readable language.

It looks like:

```
1 .super Ljava/lang/Object;
2 .method public static main([Ljava/lang/String;)V
3     .registers 2
4     sget-object v0, Ljava/lang/System;-->out:Ljava/io/PrintStream;
5     const-string    v1, "Hello World!"
6     invoke-virtual {v0, v1}, Ljava/io/PrintStream;-->println(Ljava/lang/String;)V
7     return-void
8 .end method
```

But don't worry, in most of the cases this is not the final language you're gonna read to reverse the app ;)

Given an APK, this command line will decompile it:

```
apktool d application.apk
```

Once finished, the `application` folder is created and you'll find all the output of apktool in there.


You can also use `apktool` to decompile an APK, modify it and then recompile it (like i did with the Nike+ app in order to have more debug logs for instance), but unless the other tools will fail the decompilation, it's unlikely that you'll need to read `smali` code in order to reverse the application, let's get to the other tools now ;)

jADX

The jADX suite allows you to simply load an APK and look at its Java source code. What's happening under the hood is that jADX is decompiling the APK to smali and then converting the smali back to Java. Needless to say, reading Java code is much easier than reading smali as I already mentioned :)

Once the APK is loaded, you'll see a UI like this:



One of the best features of JADX is the string/symbol search (the  button) that will allow you to search for URLs, strings, methods and whatever you want to find inside the codebase of the app.

Also, there's the `Find Usage` menu option, just highlight some symbol and right click on it, this feature will give you a list of every references to that symbol.

Dex2Jar and JD-Gui

Similar to JADX are the dex2jar and JD-GUI tools, once installed, you'll use `dex2jar` to convert an APK to a JAR file:

```
/path/to/dex2jar/d2j-dex2jar.sh application.apk
```

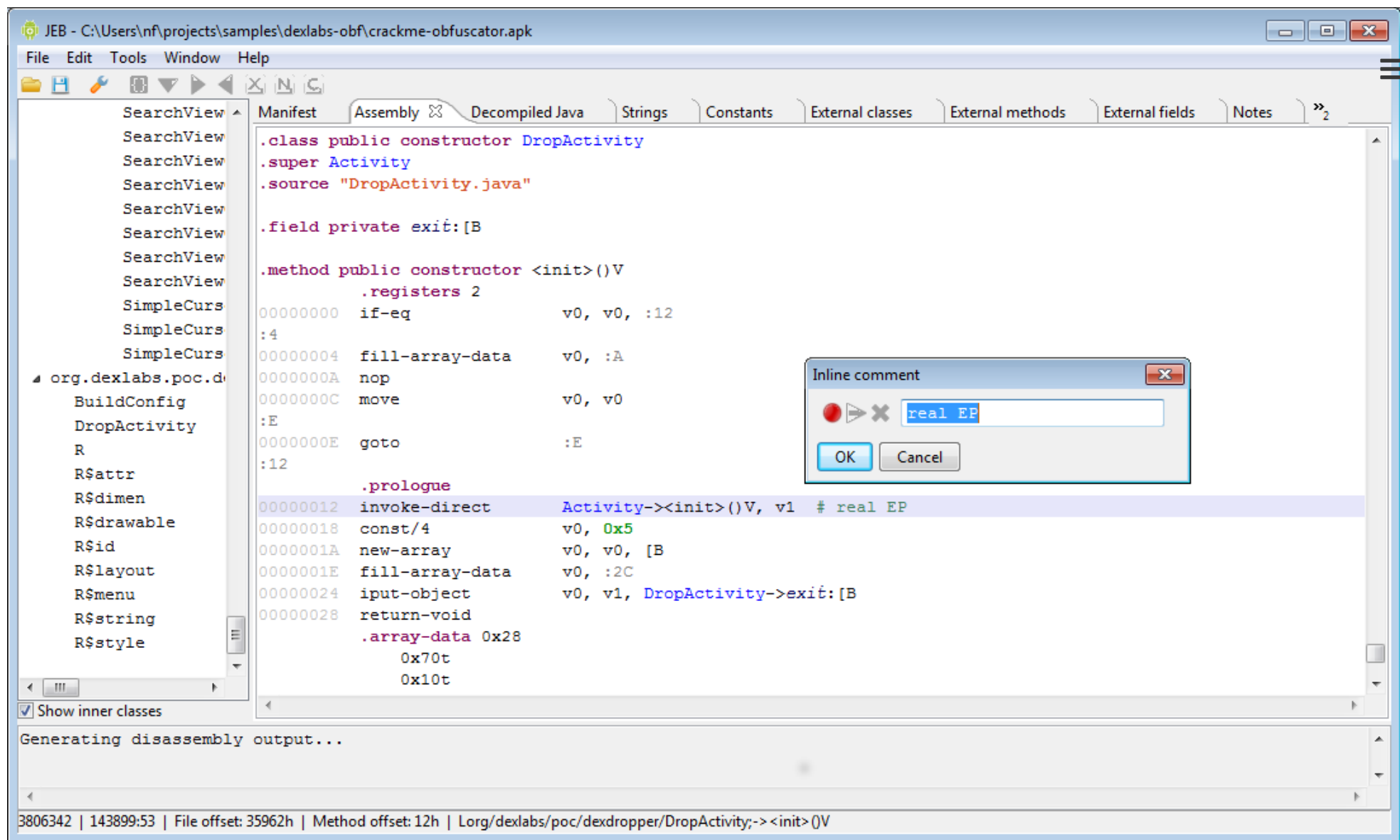
Once you have the JAR file, simply open it with JD-GUI and you'll see its Java code, pretty much like JADX:



Unfortunately JD-GUI is not as features rich as JADX, but sometimes when one tool fails you have to try another one and hope to be more lucky.

JEB

As your last resort, you can try the JEB decompiler. It's a **very** good software, but unfortunately it's not free, there's a trial version if you want to give it a shot, here's how it looks like:



JEB also features an ARM disassembler (useful when there're native libraries in the APK) and a debugger (**very** useful for dynamic analysis), but again, it's not free and it's not cheap.

Static Analysis of Native Binaries

As previously mentioned, sometimes you'll find native libraries (`.so` shared objects) inside the `lib` folder of the APK and, while reading the Java code, you'll find `native` methods declarations like the following:

```
1 public native String stringFromJNI();
```

The `native` keyword means that the method implementation is not inside the `dex` file but, instead, it's declared and executed from native code through what is called a `Java Native Interface` or JNI.

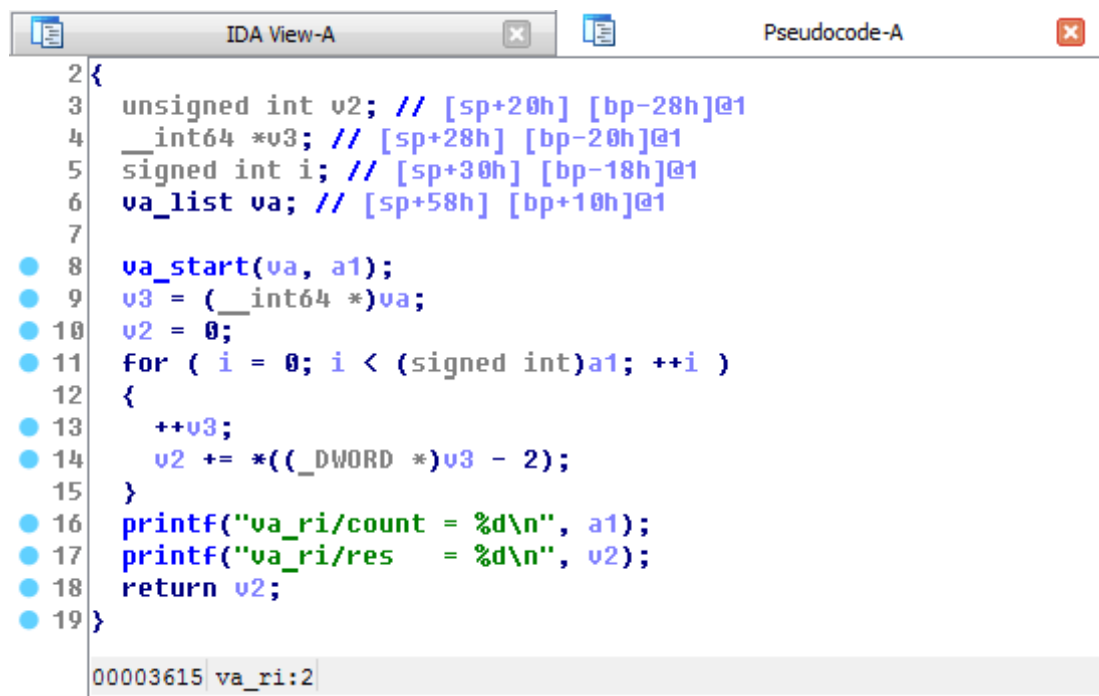
Close to native methods you'll also usually find something like this:

```
1 System.loadLibrary("hello-jni");
```

Which will tell you in which native library the method is implemented. In such cases, you will need an ARM (or x86 if there's a x86 subfolder inside the `libs` folder) disassembler in order to reverse the native object.

IDA

The very first disassembler and decompiler that every decent reverser should know about is Hex-Rays IDA which is the state of the art reversing tool for native code. Along with an IDA license, you can also buy a `decompiler` license, in which case IDA will also be able to rebuild pseudo C-like code from the assembly, allowing you to read an higher level representation of the library logic.



```
2 {
3   unsigned int v2; // [sp+20h] [bp-28h]@1
4   __int64 *v3; // [sp+28h] [bp-20h]@1
5   signed int i; // [sp+30h] [bp-18h]@1
6   va_list va; // [sp+58h] [bp+10h]@1
7
8   va_start(va, a1);
9   v3 = (__int64 *)va;
10  v2 = 0;
11  for ( i = 0; i < (signed int)a1; ++i )
12  {
13      ++v3;
14      v2 += *((_DWORD *)v3 - 2);
15  }
16  printf("va_r1/count = %d\n", a1);
17  printf("va_r1/res  = %d\n", v2);
18  return v2;
19 }
```

00003615 va_r1:2

Unfortunately IDA is a very expensive software and, unless you're reversing native stuff professionally, it's really not worth spending all those money for a single tool ... warez ... ehm ... :P

Hopper

If you're on a budget but you need to reverse native code, instead of IDA you can give Hopper a try. It's definitely not as good and complete as IDA, but it's much cheaper and will be good enough for most of the cases.

Hopper supports GNU/Linux and macOS (no Windows!) and, just like IDA, has a builtin decompiler which is quite decent considering its price:

Crypto Tools.hop

Address 0x1000e75e, Segment __TEXT, -(Crypto_ToolsAppDelegate adaptKeySize) + 174, Section __text, file offset 0xe75e

```

000000010000e70a  mov     rcx, qword [ss:rbp+var_00]
000000010000e70b  call    rcx
000000010000e70c  mov     edx, eax
000000010000e70d  imul    edx, edx, 0x4
000000010000e70e  mov     dword [ss:rbp+var_4c], edx
000000010000e70f  mov     rax, qword [ss:rbp+var_30]
000000010000e710  mov     rsi, qword [ds:0x100037d48]
000000010000e711  mov     rdi, rax
000000010000e712  call    imp__stubs_objc_msgSend
000000010000e713  mov     dword [ss:rbp+var_50], eax
000000010000e714  mov     eax, dword [ss:rbp+var_4c]
000000010000e715  mov     edx, dword [ss:rbp+var_50]
000000010000e716  cmp     eax, edx
000000010000e717  jne     loc_1000e744
000000010000e718  jmp     loc_1000e81d

loc_1000e744:
000000010000e744  mov     eax, dword [ss:rbp+var_4c]
000000010000e745  mov     ecx, dword [ss:rbp+var_50]
000000010000e746  cmp     eax, ecx
000000010000e747  jae     loc_1000e79c

000000010000e748  mov     eax, dword [ss:rbp+var_4c]
000000010000e749  mov     dword [ss:rbp+var_54], eax

loc_1000e758:
000000010000e758  mov     eax, dword [ss:rbp+var_54]
000000010000e759  mov     ecx, dword [ss:rbp+var_50]
000000010000e75a  cmp     eax, ecx
000000010000e75b  jae     loc_1000e797

000000010000e75c  lea     rax, qword [ds:cstring_0]
000000010000e75d  mov     rcx, qword [ss:rbp+var_48]
000000010000e75e  mov     rsi, qword [ds:0x100037d20]
000000010000e75f  mov     rdi, rcx
000000010000e760  mov     rdx, rax
000000010000e761  call    imp__stubs_objc_msgSend
000000010000e762  mov     qword [ss:rbp+var_40], rax
000000010000e763  mov     eax, dword [ss:rbp+var_54]
000000010000e764  add     eax, 0x4
000000010000e765  mov     dword [ss:rbp+var_54], eax
000000010000e766  jmp     loc_1000e758

loc_1000e797:
000000010000e797  jmp     loc_1000e7fb

loc_1000e79c:
000000010000e79c  mov     rax, qword [ss:rbp+var_48]
000000010000e79d  mov     ecx, dword [ss:rbp+var_50]
000000010000e79e  shr     ecx, 0x2
000000010000e79f  mov     edx, ecx
000000010000e7a0  mov     qword [ss:rbp+var_18], 0x0
000000010000e7a1  mov     qword [ss:rbp+var_20], rdx
000000010000e7a2  mov     rdx, qword [ss:rbp+var_18]
000000010000e7a3  mov     qword [ss:rbp+var_30], rdx
000000010000e7a4  mov     qword [ss:rbp+var_28], rdx
000000010000e7a5  mov     qword [ss:rbp+var_8], rdx
000000010000e7a6  mov     rdx, qword [ss:rbp+var_30]
000000010000e7a7  mov     qword [ss:rbp+var_10], rdx
000000010000e7a8  mov     rdx, qword [ss:rbp+var_10]
000000010000e7a9  mov     rsi, qword [ss:rbp+var_8]

```

Remove Hi/LO macros Remove potentially dead code


```

void -(Crypto_ToolsAppDelegate adaptKeySize)(void * self, void *
var_38 = self;
var_48 = [var_38->cipheringKey stringValue];
var_4c = [var_48 length] * 0x4;
var_50 = [var_38 getKeyBitSizeOfSelectedCipher];
if (var_4c != var_50) {
    if (var_4c < var_50) {
        for (var_54 = var_4c; var_54 < var_50; var_54++) {
            var_48 = [var_48 stringByAppendingString:
                [var_48 substringWithRange:NSMakeRange(var_54, 1)]];
        }
    }
    else {
        var_48 = [var_48 substringWithRange:NSMakeRange(0, var_4c)];
    }
    rax = var_38->cipheringKey;
    [rax setStringValue:var_48];
}
return;

```

0e70c 40 80 FF 01 89 C2 69 02 04 00 00 00 M....i....
0e718 89 55 84 48 8B 45 C8 48 8B 35 22 96 ..U.H.E.H.5".
0e724 02 00 48 89 C7 E8 82 FF 01 00 89 45 ..H.....E
0e730 00 88 45 84 8B 55 00 39 D0 0F 85 05 ..E..U.9....
0e73c 00 00 00 E9 D9 00 00 00 88 45 84 8BE..
0e748 40 00 39 C8 0F 83 4A 00 00 00 88 45 M.9.....E
0e754 B4 89 45 AC 8B 45 AC 8B 40 00 39 C8 ..E..E..M.9.
0e760 0F 83 31 00 00 00 48 8D 05 18 AC 02 ..I.....H
0e76c 00 48 8B 40 8B 48 8B 35 A8 95 02 00 .H.M.H.5....
0e778 48 89 CF 48 89 C2 E8 50 FF 01 00 48 M..H...].H
0e784 89 45 8B 8B 45 AC 05 04 00 00 89 .E..E.....
0e790 45 AC E9 C1 FF FF E9 5F 00 00 00 E.....
0e79c 48 8B 45 8B 8B 40 8B C1 E9 82 8B CA H.E..M.....
0e7a8 48 C7 45 E8 00 00 00 48 89 55 E8 H.E.....M.U.
0e7b4 48 8B 55 E8 89 55 D8 8B 55 E8 M.U..M.U..M.U..
0e7c0 48 89 55 D8 48 89 55 F8 48 8B 5D M.U..M.U..M.U..
0e7cc 48 89 55 F8 48 8B 55 F8 48 8B 75 M.U..M.U..M.U..
0e7d8 48 89 75 A0 48 89 55 98 48 8B 35 69 M..U..M.U..S.
0e7e4 95 02 00 48 8B 55 98 48 8B 40 A8 ..H.U..H.M.H
0e7f0 89 C7 E8 E9 FE 01 80 48 89 45 08 48H.E.H
0e7fc 8B 45 C8 48 8B 0D 7C 02 00 48 8B .E.H...].H.
0e808 04 00 48 8B 55 08 48 8B 35 93 94 02 ..H.U.H.5...

Dynamic Analysis

When static analysis is not enough, maybe because the application is obfuscated or the codebase is simply too big and complex to quickly isolate the routines you're interested into, you need to go dynamic. 

Dynamic analysis simply means that you'll execute the app (like we did while performing network analysis) and somehow trace into its execution using different tools, strategies and methods.


Sandboxing

Sandboxing is a black-box dynamic analysis strategy, which means you're not going to actively trace **into** the application code (like you do while debugging), but you'll execute the app into some container that will log the most relevant actions for you and will present a report at the end of the execution.

Cuckoo-Droid

Cuckoo-Droid is an Android port of the famous Cuckoo sandbox, once installed and configured, it'll give you an activity report with all the URLs the app contacted, all the DNS queries, API calls and so forth:

Dashboard
Recent
Pending
Search
Submit



Compare this analysis to...
Resubmit this sample

Quick Overview
Static Analysis
Behavioral Analysis (2)
Network Analysis (186)
Admin

Download PCAP

Hosts (8)
DNS (12)
TCP (38)
UDP (27)
HTTP/HTTPS (100)
ICMP (1)
IRC (0)
Suricata (0)
Snort (0)

HTTP & HTTPS Requests

Request	Response
<p>URL: http://mijn.ing.nl/internetbankieren/SesamLoginServlet</p> <pre> GET /internetbankieren/SesamLoginServlet HTTP/1.1 Accept: application/x-ms-application, image/jpeg, application/xaml+xml, image/gif, image/pjpeg, application/x-ms-xbap, */* Accept-Language: en-US User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Win64; x64; Trident/4.0; .NET CLR 2.0.50727; S LCC2; .NET CLR 3.5.30729; .NET CLR 3.0.30729) UA-CPU: AMD64 Accept-Encoding: gzip, deflate Host: mijn.ing.nl Connection: Keep-Alive Cookie: a7ca04= 04c4; l1 527ab7f </pre>	<pre> HTTP/1.1 200 OK Date: Sat, 28 Nov 2015 23:26:28 GMT Content-Type: text/html; charset=ISO-8859-1 Connection: keep-alive Vary: Accept-Encoding Cache-Control: no-cache Pragma: no-cache Expires: Thu, 01 Jan 1970 00:00:00 GMT Cache-Control: no-store X-Frame-Options: SAMEORIGIN Set-Cookie: _ses_ref_=; HttpOnly; HttpOnly; Path=/; Domain=.ing.nl; Secure Set-Cookie: sessiontype=mpb; Secure Set-Cookie: aac=; Expires=Sun, 27-Nov-16 23:26:27 GMT; Path=/; Domain=.ing.nl; Secure Content-Language: en-US X-Content-Type-Options: nosniff Content-Encoding: gzip Strict-Transport-Security: max-age=31622400 Set-Cookie: fc1b12d9718f Set-Cookie: f03be591cdd2 Transfer-Encoding: chunked </pre>

Joe Sandbox

The mobile Joe Sandbox is a great online service that allows you to upload an APK and get its activity report without the hassle of installing or configuring anything.

This is a sample report, as you can see the kind of information is pretty much the same as Cuckoo-Droid, plus there're a bunch of heuristics being executed in order to behaviourally correlate the sample to other known applications.


General Information

Date: 27.07.2016

Duration: 0h 2m 48s

Sample Name: meeting -EF79-..wsf

Cookbook: default.jbs

Icon: 

Filetype: 

Show File Information

Detection

MALICIOUS

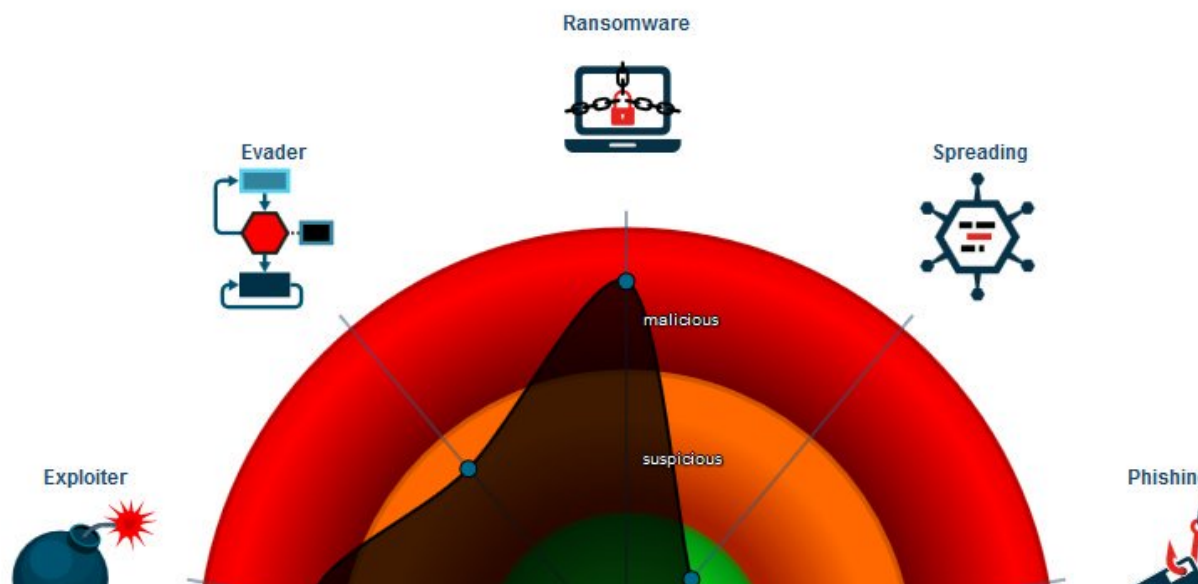
- Found **1** malicious signature
- Contacts **10** domains/IPs
- Launches **5** processes
- Drops **121** files

Signature Overview

Spam, unwanted A...	8
Networking	8
Malware Analysis S...	8
Anti Debugging	6
HIPS / PFW / Operating System ...	5
Data Obfuscation	4

Show Signature Information

Classification



Debugging

If sandboxing is not enough and you need to get deeper insights of the application behaviour, you'll need to debug it. Debugging an app, in case you don't know, means attaching to the running process with a `debugger` software, putting `breakpoints` that will allow you to stop the execution and inspect the memory state and `step` into code lines one by one in order to follow the execution graph very closely.

Enabling Debug Mode

When an application is compiled and eventually published to the Google Play Store, it's usually its `release` build you're looking at, meaning debugging has been disabled by the developer and you can't attach to it directly. In order to enable debugging again, we'll need to use `apktool` to decompile the app:

```
apktool d application.apk
```

Then you'll need to edit the `AndroidManifest.xml` generated file, adding the `android:debuggable="true"` attribute to its `application` XML node:

```
1 <?xml version="1.0" encoding="utf-8" standalone="no"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.company.appname"
3                                     platformBuildVersionCode="24"
4                                     platformBuildVersionName="7.0">
5     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
6     <uses-permission android:name="android.permission.INTERNET"/>
7
8     <application android:allowBackup="true" android:icon="@mipmap/ic_launcher"
9                 android:label="@string/app_name"
10                android:supportsRtl="true"
11                android:theme="@style/AppTheme"
12                android:debuggable="true"> <-- !!! NOTICE ME !!! -->
13
```

```
14     <activity android:name="com.company.appname.MainActivity">
15         <intent-filter>
16             <action android:name="android.intent.action.MAIN"/>
17             <category android:name="android.intent.category.LAUNCHER"/>
18         </intent-filter>
19     </activity>
20 </application>
21
22 </manifest>
```

Once you updated the manifest, let's rebuild the app:

```
apktool b -d application_path output.apk
```

Now let's resign it:

```
git clone https://github.com/appium/sign
java -jar sign/dist/signapk.jar sign/testkey.x509.pem sign/testkey.pk8 output.apk signed.apk
```

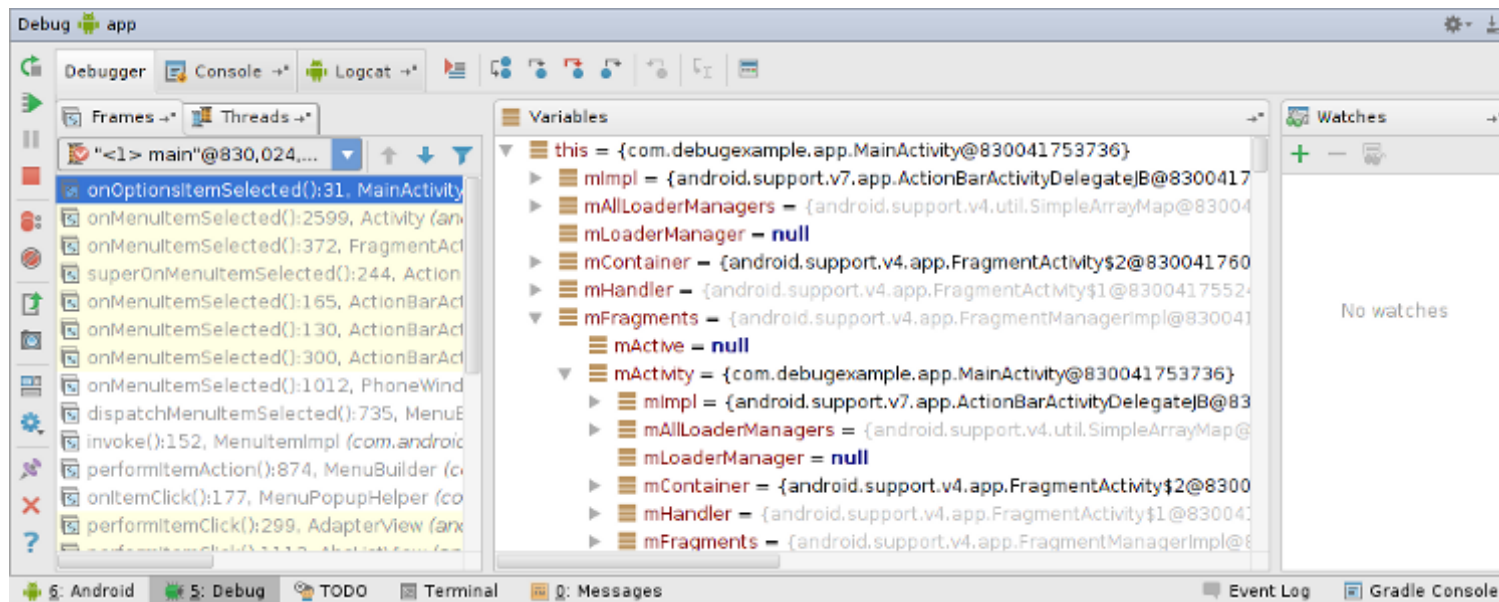
And reinstall it on the device (make sure you uninstalled the original version first):

```
adb install signed.apk
```

Now you can proceed debugging the app ^_^

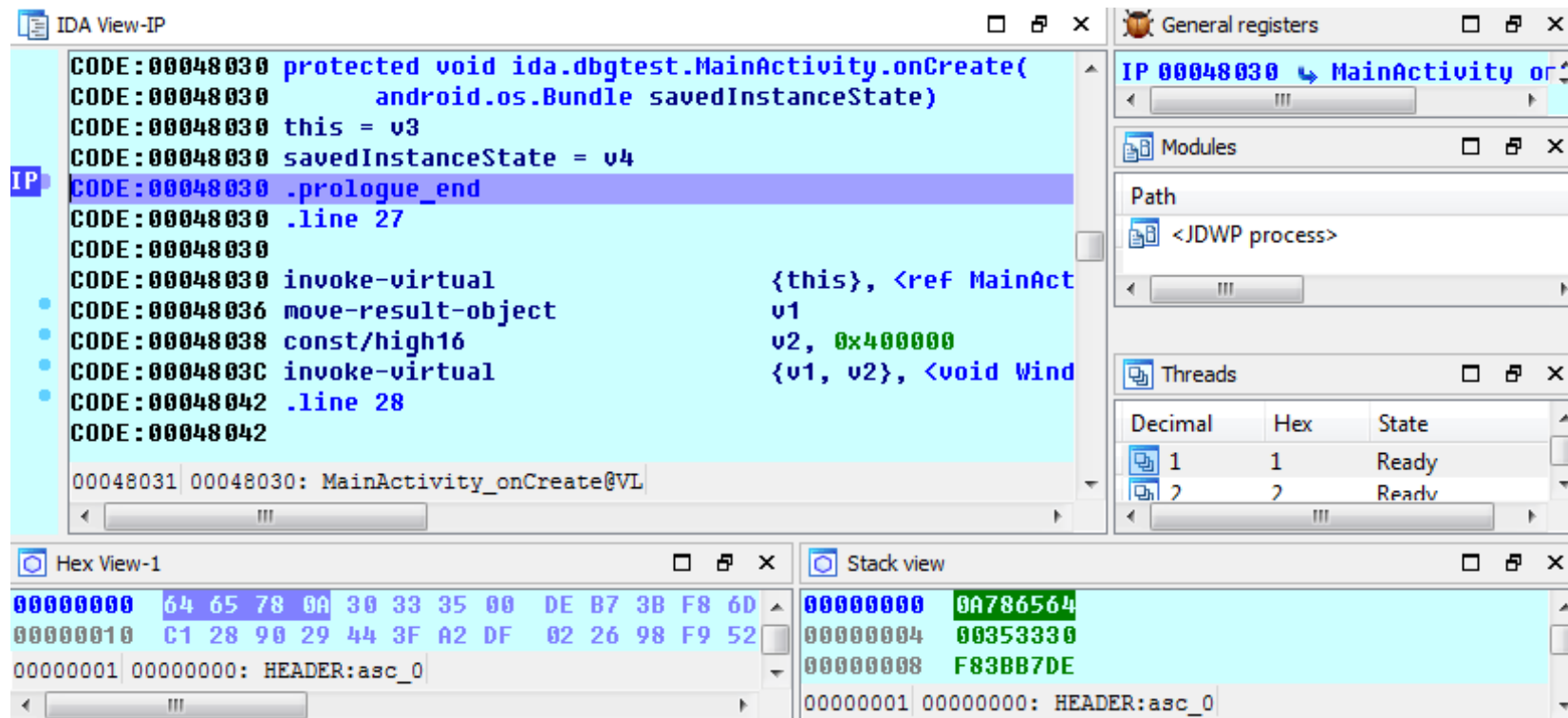
Android Studio

Android Studio is the official Android IDE, once you have debug mode enabled for your app, you can directly attach to it using this IDE and start debugging:



IDA

If you have an IDA license that supports Dalvik debugging, you can attach to a running process and step through the smali code, [this document](#) describes how to do it, but basically the idea is that you upload the ARM debugging server (a native ARM binary) on your device, you start it using `adb` and eventually you start your debugging session from IDA.



Dynamic Instrumentation

Dynamic instrumentation means that you want to modify the application behaviour at runtime and in order to do so you inject some “agent” into the app that you’ll eventually use to instrument it.

You might want to do this in order to make the app bypass some checks (for instance, if public key pinning is enforced, you might want to disable it with dynamic instrumentation in order to easily inspect the HTTPS traffic), make it show you information it’s not supposed to show (unlock “Pro” features, or debug/admin activities), etc.

Frida

Frida is a great and free tool you can use to inject a whole Javascript engine into a running process on Android, iOS and many other platforms ... but why Javascript?

Because once the engine is injected, you can instrument the app in very cool and easy ways like this:

```
1  from __future__ import print_function
2  import frida
3  import sys
4
5  # let's attach to the 'hello' process
6  session = frida.attach("hello")
7
8  # now let's create the Javascript we want to inject
9  script = session.create_script("""
10  Interceptor.attach(ptr("%s"), {
11      onEnter: function(args) {
12          send(args[0].toInt32());
13      }
14  });
15  """ % int(sys.argv[1], 16))
16
17  # this function will receive events from the js
18  def on_message(message, data):
19      print(message)
20
21  # let's start!
22  script.on('message', on_message)
23  script.load()
24  sys.stdin.read()
```

In this example, we're just inspecting some function argument, but there're hundreds of things you can do with Frida, just RTFM! and use your imagination :D



Here's a list of cool Frida resources, enjoy!

XPosed

Another option we have for instrumenting our app is using the XPosed Framework. XPosed is basically an instrumentation layer for the whole Dalvik VM which requires you to have a rooted phone in order to install it.

From XPosed wiki:

There is a process that is called "Zygote". This is the heart of the Android runtime. Every application is started as a copy ("fork") of it. This process is started by an /init.rc script when the phone is booted. The process start is done with /system/bin/app_process, which loads the needed classes and invokes the initialization methods.

This is where Xposed comes into play. When you install the framework, an extended app_process executable is copied to /system/bin. This extended startup process adds an additional jar to the classpath and calls methods from there at certain places. For instance, just after the VM has been created, even before the main method of Zygote has been called. And inside that method, we are part of Zygote and can act in its context.

The jar is located at /data/data/de.robv.android.xposed.installer/bin/XposedBridge.jar and its source code can be found [here](#). Looking at the class XposedBridge, you can see the main method. This is what I wrote about above, this gets called in the very beginning of the process. Some initializations are done there and also the modules are loaded (I will come back to module loading later).

Once you've installed XPosed on your smartphone, you can start developing your own module (again, follow the project wiki), for instance, here's an example of how you would hook the `updateClock` method of the SystemUI application in order to instrument it:

```

1 package de.robv.android.xposed.mods.tutorial;
2
3 import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;
4 import de.robv.android.xposed.IXposedHookLoadPackage;
5 import de.robv.android.xposed.XC_MethodHook;
6 import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;
7
8 public class Tutorial implements IXposedHookLoadPackage {
9     public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {
10         if (!lpparam.packageName.equals("com.android.systemui"))
11             return;
12
13         findAndHookMethod("com.android.systemui.statusbar.policy.Clock", lpparam.classLoader, "updateClock",
14             new XC_MethodHook() {
15                 @Override
16                 protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
17                     // this will be called before the clock was updated by the original method
18                 }
19                 @Override
20                 protected void afterHookedMethod(MethodHookParam param) throws Throwable {
21                     // this will be called after the clock was updated by the original method
22                 }
23             });
24     }
25 }

```

There're already a lot of [user contributed modules](#) you can use, study and modify for your own needs.

Conclusion

I hope you'll find this reference guide useful for your Android reversing adventures, keep in mind that the most important thing while reversing is not the tool you're using, but how you use it, so you'll have to learn how to choose the appropriate tool for your scenario and this is something you can only learn with experience, so enough reading and start reversing! :D

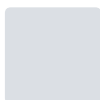




Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

**Justin Case** • 2 years ago

Good read, going to start linking people to this when they ask me how to start.

I do disagree with /assets/ not often being very important. It tends to be where packers and the light want to hide the goods.

2 ^ | ▾ • Reply • Share ›

**Simone Margaritelli** Mod → Justin Case • 2 years ago

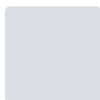
That's why I wrote "... unless of course you find inside the software functional references to such files." :D

^ | ▾ • Reply • Share ›

**stealthmuis** → Justin Case • 2 years ago • edited

Also, in case of React Native applications, /assets/index.android.bundle is worth to check :D

^ | ▾ • Reply • Share ›

**Valerio Lupi (valerino)** • 2 years ago

You forgot radare!!!! Anyway, grande come sempre!!!! :)

^ | ▾ • Reply • Share ›

**Simone Margaritelli** Mod → Valerio Lupi (valerino) • 2 years ago

No I didn't, it just never worked with ARM for me :) Moreover I tried to keep the tools list as compact as possible in order to avoid

confusion for the new comers ;) grazie! :)

1 ^ | v • Reply • Share ›



timwr → Simone Margaritelli • 2 years ago

I really recommend radare2. It works great on ARM/Android binaries.

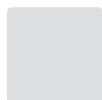
^ | v • Reply • Share ›



Ganeshkumar • 2 years ago

Wow! Very helpful for beginners

1 ^ | v • Reply • Share ›



Andreas Constantinides • 2 years ago

thanks!

^ | v • Reply • Share ›



Simone Margaritelli Mod → Andreas Constantinides • 2 years ago

you're welcome dude! :)

^ | v • Reply • Share ›



mluis • 2 years ago

Aw man.. I definitely love your posts. Awesome reading and cool subjects.

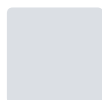
^ | v • Reply • Share ›



Simone Margaritelli Mod → mluis • 2 years ago

thanks! :)

^ | v • Reply • Share ›



Nikhil George • 2 years ago

I was trying to do network traffic analysis. I installed bettercap. My mobile device and ubuntu VM is on same wifi. I ran the command `sudo bettercap -T 192.168.1.5 --proxy --proxy-https --no-sslstrip`. It starts bettercap but no traffic is shown. Can you please help me out..?

^ | v • Reply • Share ›



yongsua1989 • 2 years ago

Dear author, may I know which programs do you recommend for kernel debugging? Thank You.

^ | v • Reply • Share ›

Simone Margaritelli Mod → yongsua1989 • 2 years ago

kgdb

^ | v • Reply • Share ›

Hackndo • 2 years ago

Thank you for this. Didn't know some tools, gonna try them out.

^ | v • Reply • Share ›

Matthias Kaiser • 2 years ago

Debugging Smali with Netbeans worked well in the past for me: <http://d-kovalenko.blogspot...>

^ | v • Reply • Share ›

Simone Margaritelli Mod → Matthias Kaiser • 2 years ago

that's cool! too bad i really can't stand NB XD

1 ^ | v • Reply • Share ›

Diego De Santiago Ruiz • 2 years ago

Thanks for write this, i really appreciate the article, i did understand much reverse android.

^ | v • Reply • Share ›

Arya Farzan • 2 years ago

There is an Android app which generates a string and passes it as a HTTP header. I want to find the algorithm behind the generation of this string, or at least to be able to execute it with Frida. I've spent months on this with no success. I've tried APKTool but I was never able to find the section that is responsible for generating this string. I then started using Frida, but I want unable to find the name of the function that generates the string, so I was never able to hook to it. I also tried Android remote debugging in Android Studio, but that also came to a dead end. After reading your post, I got the idea of using Cuckoo-Droid to find out what that function is, and then use Frida to access it. What would you say is the best way I can achieve this?

^ | v • Reply • Share ›

^ | v · Reply · Share ›

Brian · 2 years ago

The application jADX (version 0.6.1) does not work properly on Windows 10. The application hangs repeatedly when you try and perform virtually any function. I don't remember working with any software as bad as this before.

^ | v · Reply · Share ›

Esteban · a year ago

Thank you very much.

This is really good information for a start into the Reverse Engineering.

^ | v · Reply · Share ›

neverdie_py · a year ago

omg, i love this so much, thank you!!!

^ | v · Reply · Share ›

Anon2123124 · 8 months ago · edited

Great article Simone :)

Can you update the article regarding the use of bettercap with the new version 2+ ?

I don't know what is the equivalent for:

```
sudo bettercap -T 192.168.1.5 -X
```

```
sudo bettercap -T 192.168.1.5 --proxy --proxy-https --no-sslstrip
```

^ | v · Reply · Share ›

Kucing Kiki · 7 months ago

thanks..

6 ^ | v · Reply · Share ›

 [Subscribe](#)

 [Add Disqus to your site](#)

 [Disqus' Privacy Policy](#)

DISQUS

