Android Malware Reverse Engineering

Axelle Apvrille

Hack.Lu, October 2016
Hello

Welcome!

Who am I? Axelle Apvrille

▶ Security researcher at Fortinet, Fortiguard Labs
▶ Topic: malware for smart devices (phones, IoT...)
▶ Email: aapvrille at fortinet dot com
▶ Twitter: @cryptax
▶ GPG: 5CE9 C366 AFB5 4556 E981 020F 9EAA 42A0 37EC 490C
<table>
<thead>
<tr>
<th>Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Android Malware RE - Part One - 2 hours</strong></td>
</tr>
<tr>
<td>▶ Contents of an APK</td>
</tr>
<tr>
<td>▶ Static analysis</td>
</tr>
<tr>
<td>▶ Dynamic analysis</td>
</tr>
<tr>
<td><strong>Android Malware RE - Part Two - 1 hour</strong></td>
</tr>
<tr>
<td>▶ De-obfuscation</td>
</tr>
</tbody>
</table>
For the labs

Copy the contents of the USB key and pass to your neighbour!
Thanks!
**Requirements**: install either Docker or VirtualBox

https://www.docker.com/products/overview

You also need either ssh or vncviewer

https://www.virtualbox.org/wiki/Downloads
It’s a training, you are going to work :=)
And that’s me, resting, or more precisely Pico le Croco
What’s an APK?

It is a Zip!

Taken from Android/Spitmo.C!tr.spy

$ unzip criptomovil.apk
Archive:  criptomovil.apk
  inflating:  res/layout/main.xml
  inflating:  AndroidManifest.xml
extracting:  resources.arsc
extracting:  res/drawable-hdpi/icon.png
extracting:  res/drawable-ldpi/icon.png
extracting:  res/drawable-mdpi/icon.png
  inflating:  classes.dex
  inflating:  META-INF/MANIFEST.MF
  inflating:  META-INF/CERT.SF
  inflating:  META-INF/CERT.RSA
APK - Android Packages

`AndroidManifest.xml`
Binary XML

$ aapt dump xmltree ...

`AndroidManifest.xml`
XML

Resources

$ aapt dump resources ...

`resources.arsc`
Compiled resource table

assets/

`res/`
Resources

classes.dex
Dalvik bytecode

`libs/`
Libraries

$ dexdump ...

dx

javac

Java source code

$ keytool -printcert -file CERT.RSA

Certificate

META-INF/

jarsigner

Key
Apktool - all in 1 tool

https://ibotpeaches.github.io/Apktool/
Apktool and (most) other tools are already installed on the images for the lab

$ java -jar apktool.jar d YOURPACKAGE.apk -o OUTPUTDIR

- d is for decoding
- Will retrieve Android manifest, resources and smali code
Converting binary XML

Binary XML → Human readable XML

Use AXMLPrinter or newer from rednaga:

```
java -jar AXMLPrinter2.jar binary.xml
```

Alternatives:

- **aapt**: `aapt dump xmltree yourpack.apk AndroidManifest.xml`
- **Androaxml.py** from Androguard
- **Apktool**: all in one tool
How to read resources?

What if apktool does not work?

- `aapt dump resources`: works but output not excellent
- Layouts only: use AXMLPrinter, androaxml to convert binary XML to XML
Dalvik Executables (.dex)

- Dalvik Executable (DEX): similar to .class for Java
- smali means assembler in icelandic
What if apktool fails to produce smali?

- **Baksmali**
  
  `java -jar baksmali.jar -o output-dir classes.dex`

- **Androguard**:
  
  `androdd -i classes.dex -o output` or
  
  `$ androlyze -s
  d, dx = AnalyzeDex("classes.dex")
  d.create_python_export()`

- Use your favorite disassembler (if it supports it): **IDA Pro, Radare2**
You don’t like smali? Use a decompiler!

- **Androguard** embeds a good decompiler.
  
  ```python
  a, d, dx = AnalyzeAPK('sample.apk.vpk', decompiler='dad')
  d.CLASS_xxxx.METHOD_yyy.source()
  ```

- **JADX**
  
  ```bash
  jadx -d output-dir classes.dex
  ```

- **Convert to jar** using **dex2jar** and then use a Java decompiler (Krakatau, Procyon, CFR, JD, ClassyShark...)

- **Dedexer** produces .ddx files ≅ **Jasmin** w/ Dalvik opcodes

- **DED Decompiler or Dare**

- **JEB Decomplier**: not free - but excellent. Trial version exists.
Samples are located in /data
Tools are located in /opt (and subdirectories)
You have a work dir in /workshop
Password: rootpass
AdminService class, inheriting from Service. Source file name is missing:

```
.class public AdminService
.super Service
.source ""

.method static <clinit>()V
    .registers 1
    const/4 v0, 0
    sput-object v0, AdminService->c0Ic00o:Thread
    return-void
.end method
```

- Dalvik is registered based, not stack based
- Java signatures for methods: V for void, B for byte, Z for boolean...
- Dalvik instructions: const/4, sput-object...
.method public constructor <init>(I)V
    .registers 4
    .param p1, "initialCapacity"

▶ p0 is for this, p1 is first argument of method
▶ naming is not always provided!

Calls

invoke-virtual {v0, v1, p1}, L.../TinyDB;
    -> putInt(Ljava/lang/String;I)V

Means: this.putInt(v1, p1);
How to reverse Android malware?

1. First glance matters
   - Are they trying to hide something?
   - What’s the name of the package?
   - What does the certificate say?
   - Where did I find it?

2. Disassemble it
   - The code says it all!
   - Don’t be lazy 😞 and read it in depth.
   - THE CODE DOES NOT MAKE SENSE?
   - Maybe it’s heavily obfuscated or packed.

3. Still don’t understand?
   - Run it in an emulator, display logs and capture network traffic.
   - Never use your own phone.
   - Do not provide any personal data (name, IMEI, phone number...)

Good 😊 Check the assets and resources directory for Javascript or ARM executables.
Read the manifest

Taken from Android/Spitmo.C!tr.spy

- Identify the main entry point

```xml
<activity android:label="@7F040001" android:name=".MainActivity">

  <intent-filter>
    <action android:name="android.intent.action.MAIN">
      <category android:name="android.intent.category.LAUNCHER"/>
    </action>
  </intent-filter>
</activity>
```
Read the manifest

**Taken from Android/Spitmo.C!tr.spy**
- Identify the main entry point
- Background services

```xml
<service android:enabled="true" android:name=".KayService"/>
</service>
```
Read the manifest

Taken from Android/Spitmo.C!tr.spy

- Identify the main entry point
- Background services
- Receivers: called when events occur

```
<receiver android:name=".SmsReceiver">
    <intent-filter android:priority="99999">
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
        <action android:name="android.intent.action.NEW_OUTGOING_CALL"/>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```
Read the manifest

Taken from Android/Spitmo.C!tr.spy

- Identify the main entry point
- Background services
- Receivers: called when events occur
- Permissions

```
<uses-permission android:name="android.permission.READ_SMS">
</uses-permission>

<uses-permission android:name="android.permission.RECEIVE_SMS">
```
Decompiled Java source code - at a glance

Classes, fields, and methods

Bad class: localObject2 is a TelephonyManager

Dummy variable names
Who’s using this method/field?

- Good news: smali are text files. You can grep etc.
- **Androguard**: show_xref(), show_dref()
- **JEB**: Ctrl-X
- **Radare**: axt, axf

**Beware**

Inheritance, interfaces, events “break” the call tree :(
Patching an APK

Modify the smali code

1. Baksmali to get the smali
2. Modify the smali source
3. Smali to re-create the DEX
4. Zip the DEX with resources
5. Sign it (if necessary create keys before)

Patch to insert logs

```java
const-string v0, "Hello there"
const-string v6, "MY TAG: "
invoke-static {v6, v0},
    Landroid/util/Log;->v(Ljava/lang/String; Ljava/lang/String;)I
```
Lab 4: Patching a package
Dynamic analysis

- Make sure you won’t be sending data to the malware authors
- Some malware perform anti-emulator tricks
Dynamic analysis: SpyBanker in (safe) action!
Androguard: quick start

- Launch androlyze with interactive shell: androlyze -s. Python shell.
- Analyze the APK: a, d, dx = AnalyzeAPK('your.apk', decompiler='dad')
- Perform actions on the package through object a. Use completion (Tab). Example: a.get_main_activity(), a.get_receivers(), a.get_services()
- Actions on the code: use d.CLASS, then use completion (Tab). To specify a method add _METHOD and use completion. Call source() to see decompiled code, or use completion.
- Method cross references: use CLASS_xxx.METHOD_yyy.show_xref().
- Field cross references: CLASS_xxx.FIELD_yyy.show_dref()
- List used permissions: show_Permissions(dx)
**Counter anti-emulator tricks**

### IMEI

On emulator, IMEI default value is 0000000000000000. Very common check in malware.

Get the value:
- **Program**: `getDeviceId()`
- **Emulator < Android 5**: `adb shell dumpsys iphonesubinfo`
- **Emulator ≥ Android 5**: `adb shell service call iphonesubinfo code`  
  5.1.1: `code = 1`

Set the value: search for `+CGSN`

<table>
<thead>
<tr>
<th>IMEI</th>
<th>Hexadecimal</th>
<th>Value</th>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00370320</td>
<td>53 3d 00 2b 43 49 4d 49 00 33 31 30 32 36 30 30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00370330</td>
<td>30 30 30 30 30 30 30 00 2b 43 47 53 4e 00 30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00370340</td>
<td>30 30 30 30 30 30 30 30 30 30 30 30 30 30 00 2b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### IMSI

Get the value:
- **Program**: `getSubscriberId()`
- **Emulator**: same as IMEI, except service code is 7 (Android 5.1.1).

Set the value: search for `+CIMI`

### Geographic location

Common especially in Adware.
Set the value: `adb emu geo fix longitude latitude altitude`
Get the value: `adb shell dumpsys location` *(does not work on emulator)*
Lab 7: Patching the emulator
http://www.radare.org

- It works on the classes.dex. Automatic detection of Dalvik. (If not, use r2 -a dalvik file).
- List classes, methods and fields: ic, or list functions: afl
- List imports: ii
- List strings: iz (method names in there too)
- Cross references: axt (references TO this address) or axf (from)
- Search for string http: f http or / http
- Disassemble: pd LINES @ ADDR
Obfuscation...

- **Obfuscators.** Generic term. Proguard, Dexguard, Allatori,
- **Protectors.** e.g. anti-debugging, anti-emulator techniques
  ApkProtect
- **Packers.** Executable ’compressor’. Decompression stub decompresses sample *in place* (dump memory) or *on disk* (inspect /data/data for example). Pangxie, LIAPP, Bangcle
1. **Understand** how it is obfuscated and write code/scripts to de-obfuscate Identification of packers with APKiD

```
[!] APKiD 0.9.3 :: from RedNaga :: rednaga.io
[*] 2164084.apk
  |-> packer : Ijiami
[*] 2164084.apk!classes.dex
  |-> compiler : dexlib 2.x
[*] 2164084.apk!assets/ijm_lib/armeabi/libexec.so
  |-> packer : Ijiami (UPX)
[*] 2164237.apk
  |-> packer : Jiangu
[*] 2164237.apk!classes.dex
  |-> compiler : dexlib 2.x
[*] 2164332.apk!classes.dex
```
1. Understand how it is obfuscated and write code/scripts to de-obfuscate

2. **Use off-the-shelf tools that already do the work ;P**
   - d2j-decrypt-string.sh
   - DexHunter: Android 4.4.3
   - Simplify
   - JEB plugins
JEB scripts to decrypt strings
Lab 9: Using JEB Plugins
1. Understand how it is obfuscated and write code/scripts to de-obfuscate
2. Use off-the-shelf tools that already do the work ;P
3. **Modify the sample and print the de-obfuscated string/class etc.**
1. Understand how it is obfuscated and write code/scripts to de-obfuscate
2. Use off-the-shelf tools that already do the work ;P
3. Modify the sample and print the de-obfuscated string/class etc.
4. Debug the sample and set a breakpoint where you want to see the obfuscated data.
   - JEB2
   - CodeInspect
1. Understand how it is obfuscated and write code/scripts to de-obfuscate
2. Use off-the-shelf tools that already do the work ;P
3. Modify the sample and print the de-obfuscated string/class etc.
4. Debug the sample and set a breakpoint where you want to see the obfuscated data.
5. **Dump memory of the phone and search for de-obfuscated data**
   - GDB
   - kisskiss
Lab 11: Unpacking LIAPP
References

- Dalvik Opcodes
- Collection of Android tools
- Using Androguard for RE
- Emacs smali mode: Tim Strazzere
- Obfuscation in Android malware and to fight back
- Android App “Protection”
- My own publications
Thank You!

Thank you for attending! Special thanks to Ruchna Nigam, Tim Strazzere CodeInspect and JEB for providing free licenses

Please bring the USB keys back :)

Like the slides? Thanks. This is \LaTeX