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

Agenda

Android Malware RE - Part One - 2 hours

- Contents of an APK
- Static analysis
- Dynamic analysis

Android Malware RE - Part Two - 1 hour

- De-obfuscation

For the labs

Copy the contents of the USB key
and pass to your neighbour!
Thanks!

Requirements

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

Labs

It's a training, **you** are going to work :=)

And that's me, resting, or more precisely Pico le Croco

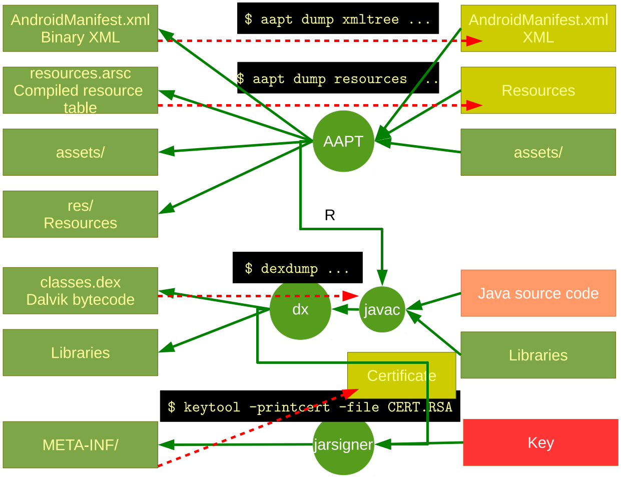
What's an APK?

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



```
$ aapt dump xmltree ...
$ aapt dump resources ...
$ dexdump ...
$ keytool -printcert -file CERT.RSA
```

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

$$\begin{tikzcd}\boxed{\text{Binary XML}} \arrow[r] & \boxed{\text{Human readable XML}}\end{tikzcd}$$

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

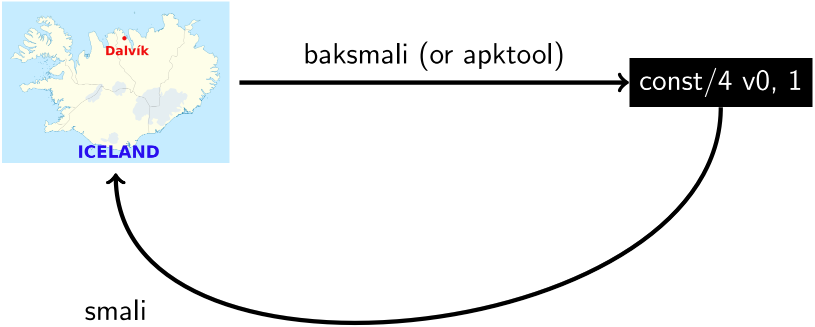
How to read resources?

$$\begin{tikzcd}\boxed{\text{resources.arsc}} \arrow[r, "\text{apktool}"] & \boxed{\text{assets, resources...}}\end{tikzcd}$$

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)



- **D**alvik **Ex**exutable (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

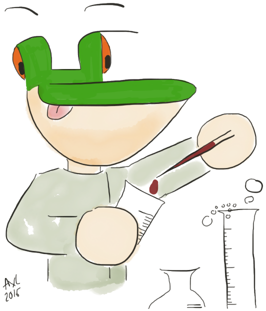
  ```
  $ androlyze -s
  d, dx = AnalyzeDex("classes.dex")
  d.create_python_export()
  ```
- Use your favorite disassembler (if it supports it): IDA Pro, Radare2

You don't like smali? Use a decompiler!

- Androguard embeds a good decompiler.

  ```
  a, d, dx = AnalyzeAPK('sample.apk.vpk',decompiler='dad')
  d.CLASS_xxxx.METHOD_yyy.source()
  ```
- JADX `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 Decompiler: not free - but excellent. Trial version exists.

Lab 1 and Lab 2



Samples are located in **/data**
Tools are located in **/opt** (and subdirectories)
You have a work dir in /workshop
Password: **rootpass**

Understanding Smali

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...`

Understanding smali 2/2

```
.method public constructor <init>(I)V
          .registers 4
          .param p1, "initialCapacity"
```

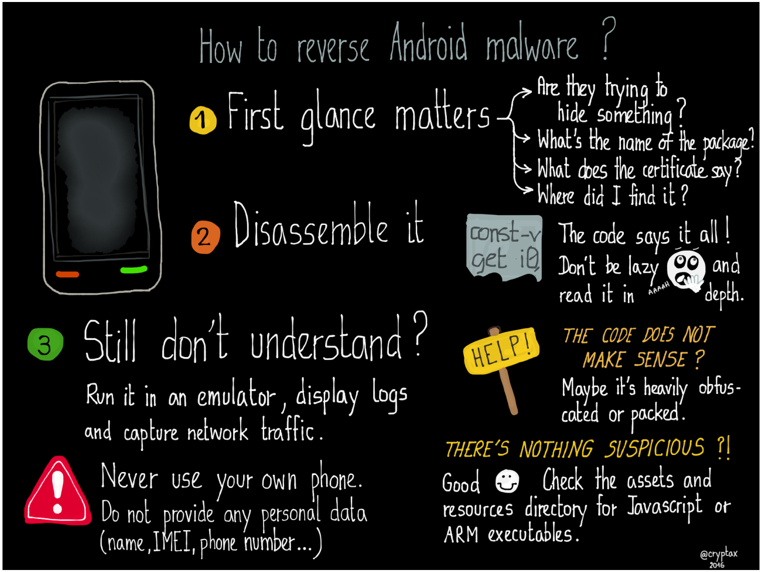
- p0 is for this, p1 is first argument of method
- naming is not always provided!

Calls

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

Means: `this.putInt(v1, p1);`

Guidelines



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

```
const-v
get i0
```

The code says it all ! Don't be lazy and read it in AAAAH depth.

3 Still don't understand ?

Run it in an emulator, display logs and capture network traffic.

THE CODE DOES NOT MAKE SENSE ?

Maybe it's heavily obfuscated or packed.

Never use your own phone.
Do not provide any personal data (name, IMEI, phone number...)

THERE'S NOTHING SUSPICIOUS ?!

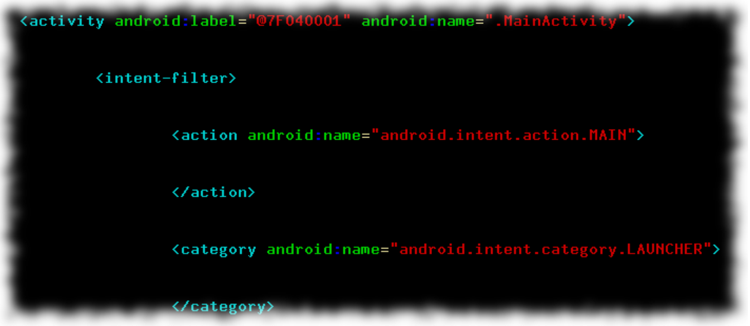
Good Check the assets and resources directory for Javascript or ARM executables.

@cryptax 2016

Read the manifest

Taken from Android/Spitmo.C!tr.spy

- Identify the main entry point

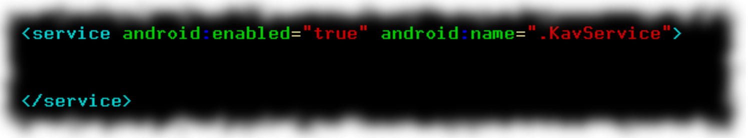


```
<activity android:label="@7F040001" android:name=".MainActivity">

        <intent-filter>

                <action android:name="android.intent.action.MAIN">

                </action>

                <category android:name="android.intent.category.LAUNCHER">

                </category>
```

Read the manifest

Taken from Android/Spitmo.C!tr.spy

- Identify the main entry point
- Background services

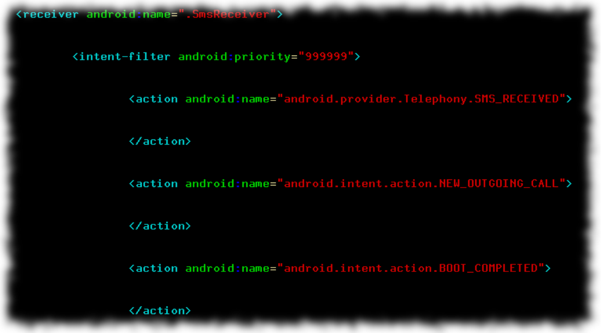


```
<service android:enabled="true" android:name=".KavService">
[redacted]
</service>
```

Read the manifest

Taken from Android/Spitmo.C!tr.spy

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

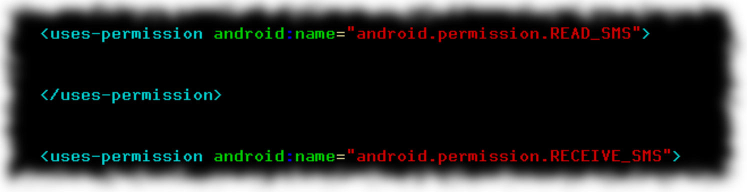


```
<receiver android:name=".SmsReceiver">

        <intent-filter android:priority="999999">

                <action android:name="android.provider.Telephony.SMS_RECEIVED">

                </action>

                <action android:name="android.intent.action.NEW_OUTGOING_CALL">

                </action>

                <action android:name="android.intent.action.BOOT_COMPLETED">

                </action>
```

Read the manifest

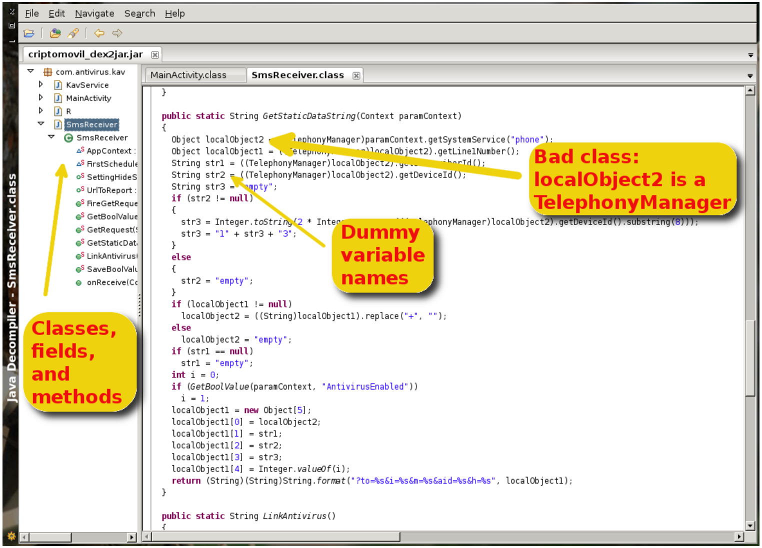
Taken from Android/Spitmo.C!tr.spy

- 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



```
}

public static String GetStaticDataString(Context paramContext)
{
  Object localObject2 = [illegible]ephonyManager)paramContext.getSystemService("phone");
  Object localObject1 = ([illegible]elephon[illegible])localObject2).getLine1Number();
  String str1 = ((TelephonyManager)localObject2).ge[illegible]iberId();
  String str2 = ((TelephonyManager)localObject2).getDeviceId();
  String str3 = [illegible]mpty";
  if (str2 != null)
  {
    str3 = Integer.toString(2 * Integ[illegible]ephonyManager)localObject2).getDeviceId().substring(8)));
    str3 = "1" + str3 + "3";
  }
  else
  {
    str2 = "empty";
  }
  if (localObject1 != null)
    localObject2 = ((String)localObject1).replace("+", "");
  else
    localObject2 = "empty";
  if (str1 == null)
    str1 = "empty";
  int i = 0;
  if (GetBoolValue(paramContext, "AntivirusEnabled"))
    i = 1;
  localObject1 = new Object[5];
  localObject1[0] = localObject2;
  localObject1[1] = str1;
  localObject1[2] = str2;
  localObject1[3] = str3;
  localObject1[4] = Integer.valueOf(i);
  return (String)(String)String.format("?to=%s&i=%s&m=%s&aid=%s&h=%s", localObject1);
}

public static String LinkAntivirus()
{
```

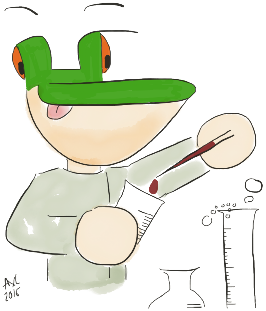
Who's using this method/field?

- 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 :(

Lab 3: Static analysis



Patching an APK

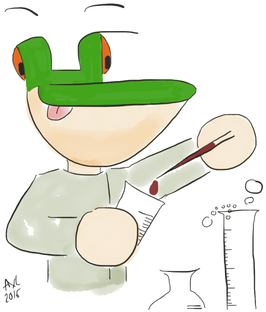
Modify the smali code

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

```
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
```

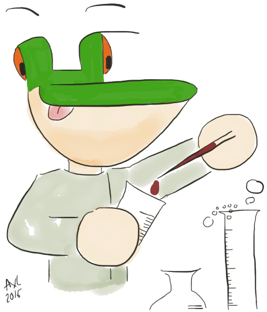
Lab 4: Patching a package



Dynamic analysis

- Make sure you won't be sending data to the malware authors
- Some malware perform anti-emulator tricks

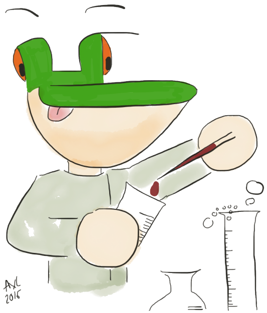
Dynamic analysis: SpyBanker in (safe) action!



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)`

Lab 6: Using Androguard



Counter anti-emulator tricks

IMEI

On emulator, IMEI default value is 000000000000000.
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**

```
00370320  53 3d 00 2b 43 49 4d 49  00 33 31 30 32 36 30 30  |S=.+CIMI.3102600|
00370330  30 30 30 30 30 30 30 30  00 2b 43 47 53 4e 00 30  |00000000.+CGSN.0|
00370340  30 30 30 30 30 30 30 30  30 30 30 30 30 30 00 2b  |00000000000000.+|
```

More anti-emulator tricks (and solutions)

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**

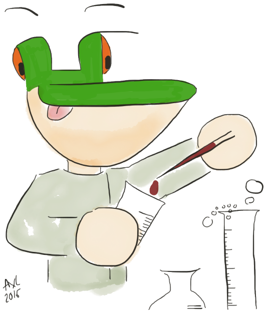
Geographic location

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

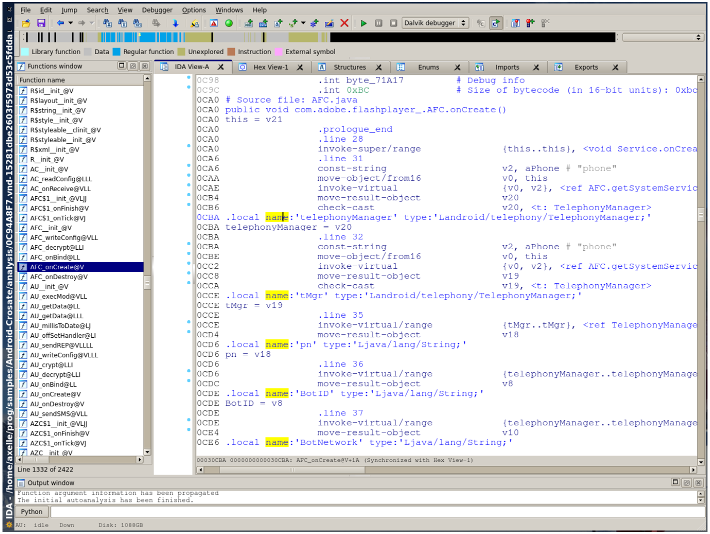


Dalvik disassembly with Radare

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`

Dalvik in IDA Pro


IDA - /home/axelle/prog/samples/Android-Crosate/analysis/0C94A8F7.vnd-15281dbe2603f5973d53c5fdda
File Edit Jump Search View Debugger Options Windows Help
Dalvik debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions window
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
Output window
Python

```
Function name
R$id__init_@V
R$layout__init_@V
R$string__init_@V
R$style__init_@V
R$styleable__clinit_@V
R$styleable__init_@V
R$xml__init_@V
R__init_@V
AC__init_@V
AC_readConfig@LLL
AC_onReceive@VLL
AFC$1__init_@VLJJ
AFC$1_onFinish@V
AFC$1_onTick@VJ
AFC__init_@V
AFC_writeConfig@VLL
AFC_decrypt@LLI
AFC_onBind@LL
AFC_onCreate@V
AFC_onDestroy@V
AU__init_@V
AU_execMod@VLL
AU_getData@LL
AU_getData@LLL
AU_millisToDate@LJ
AU_offSetHandler@LI
AU_sendREP@VLLLL
AU_writeConfig@VLLL
AU_crypt@LLI
AU_decrypt@LLI
AU_onBind@LL
AU_onCreate@V
AU_onDestroy@V
AU_sendSMS@VLL
AZC$1__init_@VLJJ
AZC$1_onFinish@V
AZC$1_onTick@VJ
AZC__init_@V
Line 1332 of 2422
```

```
0C98                 .int byte_71A17          # Debug info
0C9C                 .int 0xBC                # Size of bytecode (in 16-bit units): 0xbc
0CA0 # Source file: AFC.java
0CA0 public void com.adobe.flashplayer_.AFC.onCreate()
0CA0 this = v21
0CA0                 .prologue_end
0CA0                 .line 28
0CA0                 invoke-super/range                {this..this}, <void Service.onCrea
0CA6                 .line 31
0CA6                 const-string                      v2, aPhone # "phone"
0CAA                 move-object/from16                v0, this
0CAE                 invoke-virtual                    {v0, v2}, <ref AFC.getSystemServic
0CB4                 move-result-object                v20
0CB6                 check-cast                        v20, <t: TelephonyManager>
0CBA .local name:'telephonyManager' type:'Landroid/telephony/TelephonyManager;'
0CBA telephonyManager = v20
0CBA                 .line 32
0CBA                 const-string                      v2, aPhone # "phone"
0CBE                 move-object/from16                v0, this
0CC2                 invoke-virtual                    {v0, v2}, <ref AFC.getSystemServic
0CC8                 move-result-object                v19
0CCA                 check-cast                        v19, <t: TelephonyManager>
0CCE .local name:'tMgr' type:'Landroid/telephony/TelephonyManager;'
0CCE tMgr = v19
0CCE                 .line 35
0CCE                 invoke-virtual/range              {tMgr..tMgr}, <ref TelephonyManage
0CD4                 move-result-object                v18
0CD6 .local name:'pn' type:'Ljava/lang/String;'
0CD6 pn = v18
0CD6                 .line 36
0CD6                 invoke-virtual/range              {telephonyManager..telephonyManage
0CDC                 move-result-object                v8
0CDE .local name:'BotID' type:'Ljava/lang/String;'
0CDE BotID = v8
0CDE                 .line 37
0CDE                 invoke-virtual/range              {telephonyManager..telephonyManage
0CE4                 move-result-object                v10
0CE6 .local name:'BotNetwork' type:'Ljava/lang/String;'
00030CBA 0000000000030CBA: AFC_onCreate@V+1A (Synchronized with Hex View-1)
```

```
Function argument information has been propagated
The initial autoanalysis has been finished.
AU:  idle   Down       Disk: 1088GB
```

Obfuscation...

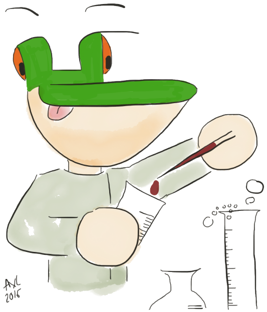
- **Obfuscators**. Generic term. Proguard, Dexguard, Allatori,
- **Protectors**. e.g. anti-debugging, anti-emulator techniques ApkProtect
- **Packers**. Executable 'compressor'. Decompression stub decompresses sample *in place* (dump memory) or *on disk* (inspect /data/data for example). Pangxie, LIAPP, Bangcle

Solutions to obfuscated malware 1/5

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
```

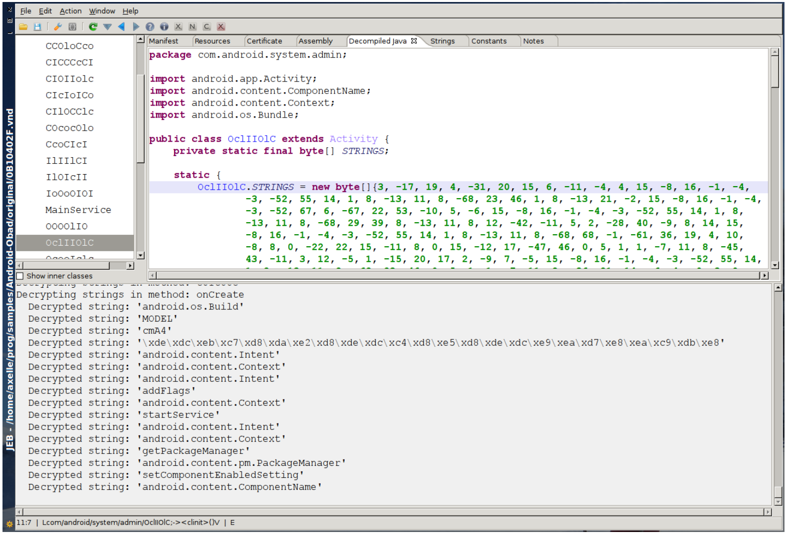
Lab 8: De-obfuscating Obad strings



Solutions to obfuscated malware 2/5

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

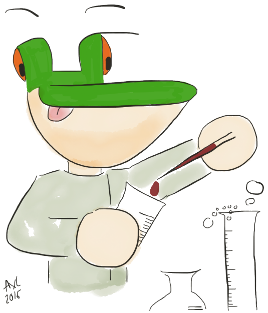
JEB scripts to decrypt strings


JEB - /home/axelle/prog/samples/Android-Obad/original/0B10402F.vnd
File Edit Action Window Help
Manifest Resources Certificate Assembly Decompiled Java Strings Constants Notes
CCOloCco
CICCCcCI
CIOIIolc
CIcIoICo
CIlOCClc
COcocOlo
CcoCIcI
IlIIlCI
IlOIcII
IoOoOIOI
MainService
OOOOlIO
OclIIOlC
Show inner classes
11:7 | Lcom/android/system/admin/OclIIOlC;-><clinit>()V | E

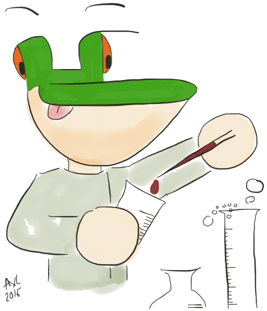
```
package com.android.system.admin;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.os.Bundle;

public class OclIIOlC extends Activity {
    private static final byte[] STRINGS;

    static {
        OclIIOlC.STRINGS = new byte[]{3, -17, 19, 4, -31, 20, 15, 6, -11, -4, 4, 15, -8, 16, -1, -4,
                -3, -52, 55, 14, 1, 8, -13, 11, 8, -68, 23, 46, 1, 8, -13, 21, -2, 15, -8, 16, -1, -4,
                -3, -52, 67, 6, -67, 22, 53, -10, 5, -6, 15, -8, 16, -1, -4, -3, -52, 55, 14, 1, 8,
                -13, 11, 8, -68, 29, 39, 8, -13, 11, 8, 12, -42, -11, 5, 2, -28, 40, -9, 8, 14, 15,
                -8, 16, -1, -4, -3, -52, 55, 14, 1, 8, -13, 11, 8, -68, 68, -1, -61, 36, 19, 4, 10,
                -8, 8, 0, -22, 22, 15, -11, 8, 0, 15, -12, 17, -47, 46, 0, 5, 1, 1, -7, 11, 8, -45,
                43, -11, 3, 12, -5, 1, -15, 20, 17, 2, -9, 7, -5, 15, -8, 16, -1, -4, -3, -52, 55, 14,
```

```
Decrypting strings in method: onCreate
  Decrypted string: 'android.os.Build'
  Decrypted string: 'MODEL'
  Decrypted string: 'cmA4'
  Decrypted string: '\xde\xdc\xeb\xc7\xd8\xda\xe2\xd8\xde\xdc\xc4\xd8\xe5\xd8\xde\xdc\xe9\xea\xd7\xe8\xea\xc9\xdb\xe8'
  Decrypted string: 'android.content.Intent'
  Decrypted string: 'android.content.Context'
  Decrypted string: 'android.content.Intent'
  Decrypted string: 'addFlags'
  Decrypted string: 'android.content.Context'
  Decrypted string: 'startService'
  Decrypted string: 'android.content.Intent'
  Decrypted string: 'android.content.Context'
  Decrypted string: 'getPackageManager'
  Decrypted string: 'android.content.pm.PackageManager'
  Decrypted string: 'setComponentEnabledSetting'
  Decrypted string: 'android.content.ComponentName'
```

Lab 9: Using JEB Plugins



Lab 10: Unpacking Pangxie



Solutions to obfuscated malware 3/5

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.**

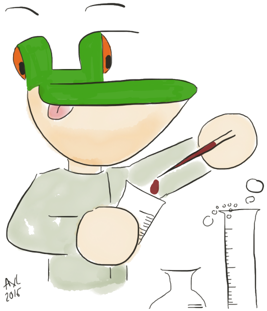
Solutions to obfuscated malware 4/5

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

Solutions to obfuscated malware 5/5

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

The end

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