My favorites ▾ | Sign in

androguard

Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)

Search projects

Project Home Downloads Wiki Issues Source

Search Current pages for Search

RE

Reverse Engineering Tutorial of Android Apps

Featured

Updated Dec 5, 2012 by anthony....@gmail.com

Reverse Engineering

This wiki page is a tutorial if you would like to analyze an android application with Androguard :) It is always update with the latest functions !

Interactive Analysis

One of the main tools in Androguard to analyze an application is *androlyze.py* because it's an interactive tool (it is a classical ipython shell, with high level commands) :

```
desnos@t0t0:/androguard$ ./androlyze.py -s
Androlyze version 1.5

In [1]:
```

In the next sections, we will see what is the main commands in *androlyze.py*, but you can use the framework to build your own tools, and to have an automatic analysis tool.

Disassemble an android application

AnalyzeAPK will do all the interesting job automaticaly for you, but if you are curious, or if you want to create your own tool, you must begin by opening the APK file:

```
a = APK("/home/t0t0/t0t0.apk")
```

and disassemble the classes.dex file, by getting the content of this file directly with get_dex method:

```
d = DalvikVMFormat( a.get_dex() )
```

but if you would like to analyse another DEX file in your APK file, you can use the get_file method:

```
d = DalvikVMFormat( a.get_file("YOUR_DEX_FILE") )
```

and analyze the classes to get the control flow graph, setup references, and create xref/dref:

```
dx = VMAnalysis( d )
gx = GVMAnalysis( dx, None )
d.set_vmanalysis( dx )
d.set_gvmanalysis( gx )

d.create_xref()
d.create_dref()
```

and finally extend python namespaces with classes/methods/fields names, and setup the display:

```
d.create_python_export()
set_pretty_show( 1 )
```

So, you can call this function directly by giving a string which represents the filename of your application:

```
In [1]: a, d, dx = AnalyzeAPK("./apks/malwares/rootsmart/suspect.apk")
In [2]: print a, d, dx
<apk.APK instance at 0x962dfcc> <dvm.DalvikVMFormat object at 0x9658d8c> <analysis.VMAnalysis instance at 0xb3b55cc
```

You will have 3 objects which represent the APK, the classes.dex, and the analyzed classes.dex. For each object, you can access directly to various fields and methods, check the documentation API.

If you have only the dex file you can call *AnalyzeDex* function:

```
In [1]: d, dx = AnalyzeDex("./apks/classes.dex")
```

Raw buffer

If you have not the file on your disk drive, you can specify the *raw* option if you wish to analyse a python buffer. For example, lot of android applications embedded other android apps:

```
In [5]: a, d, dx = AnalyzeAPK("./apks/malwares/foncy.b/213e042b3d5b489467c5a461ffdd2e38edaa0c74957f0b1a0708027e66080890

In [6]: a1, d1, dx1 = AnalyzeAPK( a.get_file("assets/border01.png"), raw=True)
```

Decompile an android application

You can add a parameter to *AnalyzeAPK*/*AnalyzeDex* in order to decompile automatically also the application by using one of the available decompiler (DAD is installed by default !):

```
In [1]: a, d, dx = AnalyzeAPK("./apks/malwares/rootsmart/suspect.apk", decompiler="dex2jad")
In [1]: a, d, dx = AnalyzeAPK("./apks/malwares/rootsmart/suspect.apk", decompiler="ded")
In [1]: a, d, dx = AnalyzeAPK("./apks/malwares/rootsmart/suspect.apk", decompiler="dad")
```

The main difference is that you are now able to use the *source* method to display the source code of a class or a method.

You must install these decompilers on a specific path by following the previous wiki page, but it is possible to have errors with such decompilers. Please don't report us a problem with a decompiler because they are close source, expect if you use DAD :)

Display

Moreover we have extend the python namespace to have an access to all classes/methods/fields. By using the completion "tab" you can have access to all elements and for each important object you will have a *show* or *pretty_show* methods which display it.

During the session, if an output doesn't fit in your terminal, you can use the page command of ipython:

```
In [4]: a, d, dx = AnalyzeAPK("./apks/com.rovio.angrybirds-2020.apk")

In [5]: z = d.get_strings()

In [6]: %page z
```

Colors

APK

You can display the APK object with the *show* method, and you will have information about files, permissions and differents entry points(activities, services...).

By default, Androguard used the zipfile module of python. But you can have problems with malformated zip files with python < 2.7, so you can used chilkat. Otherwise, if you can use the internal zipfile module from python which has been patched (see the zipmodule option of the APK class).

Information

```
In [4]: a.show()
FILES :
        META-INF/MANIFEST.MF ASCII text, with CRLF line terminators 4d14f203
        META-INF/SHIYI.SF ASCII text, with CRLF line terminators -51be4c70
        META-INF/SHIYI.RSA data -77df883f
        [....]
PERMISSIONS :  {'android.permission.READ_SYNC_SETTINGS': ['normal', 'read sync settings', 'Allows an application to rea
ACTIVITIES :  ['com.bwx.bequick.EulaActivity', 'com.bwx.bequick.ShowSettingsActivity', 'com.bwx.bequick.DialogSettingsA
SERVICES :  ['com.google.android.smart.McbainServicce']
RECEIVERS :  ['com.bwx.bequick.flashlight.LedFlashlightReceiver', 'com.bwx.bequick.receivers.StatusBarIntegrationReceiv
PROVIDERS :  []
```

You can get the content of each file in the APK file :

```
In [10]: a.get_file("res/raw/data_2")
Out[10]: '\x94QH\x17\x96\xf8l\xbd\xf9\xfdr\r~a\x14w!"wKk\xa9\'\xd3*\x1e\xd7g\x91n \x17'

In [11]: len(a.get_file("classes.dex"))
Out[11]: 200832
```

and you have various methods to get more information:

```
In [5]: a.get_package()
Out[5]: u'com.google.android.smart'

In [6]: a.get_files_crc32()
Out[6]:
{'AndroidManifest.xml': -1935393453,
 'META-INF/MANIFEST.MF': 1293218307,
 'META-INF/SHIYI.RSA': -2011138111,
```

```
 'META-INF/SHIYI.SF': -1371425904,
 [...]
}

In [7]: a.get_target_sdk_version()
Out[7]: u'8'
```

Certificate

DEX

The DEX object (DalvikVMformat class) represents the "classes.dex" file, so it is possible to have an access to each attribute of the format. But we are interesting about the classes and methods :) So if you would like to navigate to each classes/methods/fields you can use the completion, it is really easier.

```
In [3]: d.CLA
Display all 152 possibilities? (y or n)
```

```
In [3]: d.CLASS_Lcom_google_android_smart_s
Out[3]: <dvm.ClassItem instance at 0x9f16dcc>
```

and in a class, you can access to all methods/fields (the format is name+descriptor if there is identical names, otherwise you have only the name):

```
In [4]: d.CLASS_Lcom_google_android_smart_s.ME
d.CLASS_Lcom_google_android_smart_s.METHOD_a_JV
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Landroid_content_ContextLcom_google_android_smart_s
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Landroid_content_ContextLjava_lang_StringZ
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Landroid_content_IntentV
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_io_InputStreamLjava_lang_StringLjava_lang_StringIZ
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_io_InputStreamLjava_lang_StringV
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_String
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringLjava_lang_StringV
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV
d.CLASS_Lcom_google_android_smart_s.METHOD_b_JV
d.CLASS_Lcom_google_android_smart_s.METHOD_b_Ljava_lang_String
d.CLASS_Lcom_google_android_smart_s.METHOD_c_JV
d.CLASS_Lcom_google_android_smart_s.METHOD_c_Ljava_lang_String
d.CLASS_Lcom_google_android_smart_s.METHOD_d
d.CLASS_Lcom_google_android_smart_s.METHOD_e
d.CLASS_Lcom_google_android_smart_s.METHOD_f
d.CLASS_Lcom_google_android_smart_s.METHOD_g
d.CLASS_Lcom_google_android_smart_s.METHOD_h
d.CLASS_Lcom_google_android_smart_s.METHOD_i
d.CLASS_Lcom_google_android_smart_s.METHOD_init
d.CLASS_Lcom_google_android_smart_s.METHOD_j
d.CLASS_Lcom_google_android_smart_s.METHOD_k
d.CLASS_Lcom_google_android_smart_s.METHOD_l
d.CLASS_Lcom_google_android_smart_s.METHOD_m
d.CLASS_Lcom_google_android_smart_s.METHOD_n
```

```
In [4]: d.CLASS_Lcom_google_android_smart_s.F
d.CLASS_Lcom_google_android_smart_s.FIELD_a  d.CLASS_Lcom_google_android_smart_s.FIELD_c  d.CLASS_Lcom_google_android_sı
d.CLASS_Lcom_google_android_smart_s.FIELD_b  d.CLASS_Lcom_google_android_smart_s.FIELD_d  d.CLASS_Lcom_google_android_sı
```

If you would like to have more information about a method, you can access it directly:

```
In [6]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV
Out[6]: <dvm.EncodedMethod instance at 0xa7e55ec>

In [7]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom                    d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFto                      d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV._EncodedMethod__CM          d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV._EncodedMethod__method_idx  d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV._EncodedMethod__offset      d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.__class__                   d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.__doc__                     d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.__init__                    d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.__module__                  d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV._class_name                 d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV._code                       d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV._name                       d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV._proto                      d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.access_flags                d.CLASS_Lcom_google_android
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.adjust_idx                  d.CLASS_Lcom_google_android
```

```
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.code_off                    d.CLASS_Lcom_google_android
```

Most of the time, you would like to display the method, for that you can use the "show"/"pretty_show" methods:

```
In [5]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.pretty_show()
        METHOD access_flags=17 (Lcom/google/android/smart/s; a,(Ljava/lang/String;)V)
********************************************************************************
DALVIK_CODE :
        REGISTERS_SIZE 0x4
        INS_SIZE 0x2
        OUTS_SIZE 0x2
        TRIES_SIZE 0x0
        DEBUG_INFO_OFF 0x0
        INSNS_SIZE 0x15

a-BB@0x0 :
        0(0) new-instance v0 , [type@ 305 Ljava/io/File;]
        1(4) invoke-direct v0 , v3 , [meth@ 1407 Ljava/io/File; (Ljava/lang/String;) V <init>]
        2(a) invoke-virtual v0 , [meth@ 1409 Ljava/io/File; () Z exists]
        3(10) move-result v1
        4(12) if-eqz v1 , [+ 11] [ a-BB@0x16  a-BB@0x28 ]

a-BB@0x16 :
        5(16) iget-object v1 , v2 , [field@ 906 Lcom/google/android/smart/s; Landroid/content/Context; c]
        6(1a) invoke-virtual v0 , [meth@ 1411 Ljava/io/File; () Ljava/lang/String; getName]
        7(20) move-result-object v0
        8(22) invoke-virtual v1 , v0 , [meth@ 310 Landroid/content/Context; (Ljava/lang/String;) Z deleteFile] [ a-BB@0

a-BB@0x28 :
        9(28) return-void

********************************************************************************
F: Lcom/google/android/smart/u; a ()V ['6a']
F: Lcom/google/android/smart/f; a ()V ['1de', '1ec', '2da', '2e8', '426', '434', '4f2', '500', '5be', '5cc']
F: Lcom/google/android/smart/k; a ()V ['1a6']
F: Lcom/google/android/smart/l; a ()V ['18a', '1aa', '1ca']
```

The latest part of the display is the methods references. "F" means where this method is called, and "T" means which methods is called in this method. The hexadecimal values at the end represents where is the "call".

You can also display the java source code (with colors by using pygments module) if you have used a decompiler (by default, DAD will be used !):

```
In [18]: d.CLASS_Lcom_google_android_smart_BcbootReceivecr.METHOD_onReceive.source()
public void onReceive(Context context, Intent intent)
    {
        if(!s.a(context).a.d())
        {
            Intent intent1 = new Intent(context, com/google/android/smart/McbainServicce);
            intent1.setAction("action.boot");
            intent1.setFlags(0x10000000);
            context.startService(intent1);
        }

    }
```

Modification of DEX file

Permissions

You can get which permissions are used in the APK:

```
In [19]: a.get_permissions()
Out[19]:
['android.permission.ACCESS_WIFI_STATE',
 'android.permission.CHANGE_WIFI_STATE',
 'android.permission.BLUETOOTH',
 [...]
]
```

and we have a special method, *show_Permissions*, which can show where a specific permission is used with the analyzed dex (and what it is the used API):

```
In [20]: show_Permissions(dx)
WRITE_SETTINGS :
Lcom/bwx/bequick/handlers/AirplaneModeSettingHandler; setAirMode (Z)V (@setAirMode-BB@0x16-0x16)  ---> Landroid/provide
[...]
```

```
ACCESS_FINE_LOCATION :
Lcom/bwx/bequick/handlers/GpsSettingHandler; activate (Lcom/bwx/bequick/MainSettingsActivity;)V (@activate-BB@0x0-0x18)
READ_PHONE_STATE :
Lcom/google/android/smart/g; a ()V (@a-BB@0x16-0x1e)  ---> Landroid/telephony/TelephonyManager; getSimSerialNumber ()Lj
Lcom/google/android/smart/g; a (Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/S
Lcom/google/android/smart/g; a (Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/S
Lcom/google/android/smart/g; a (Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/S
[...]
```

You can do it yourself with analyzed dex, and the *get_permissions* method:

```
In [22]: p = dx.get_permissions( [] )

In [23]: p["READ_PHONE_STATE"]
Out[23]:
[<analysis.PathP instance at 0xcf9978c>,
 <analysis.PathP instance at 0xcf9f64c>,
 <analysis.PathP instance at 0xcf9ff8c>,
 <analysis.PathP instance at 0xcfa106c>]
```

You will have a list of *PathP* objects which represent where a specific method is called:

```
In [24]: z = p["READ_PHONE_STATE"][0]
In [33]: z.get_method().get_class_name(), z.get_method().get_name(), z.get_method().get_descriptor()
Out[33]: ('Lcom/google/android/smart/g;', 'a', '()V')

In [34]: z.class_name, z.name, z.descriptor
Out[34]:
('Landroid/telephony/TelephonyManager;',
 'getSimSerialNumber',
  '()Ljava/lang/String;')
In [36]: z.get_offset()
Out[36]: 30
```

It's possible to use the *show_Path* method also:

```
In [38]: show_Paths(d, i)
Lcom/google/android/smart/g; a ()V (@a-BB@0x16-0x1e)  ---> Landroid/telephony/TelephonyManager; getSimSerialNumber ()Lj

In [39]: show_Paths(d, p["READ_PHONE_STATE"])
Lcom/google/android/smart/g; a ()V (@a-BB@0x16-0x1e)  ---> Landroid/telephony/TelephonyManager; getSimSerialNumber ()Lj
Lcom/google/android/smart/g; a (Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/S
Lcom/google/android/smart/g; a (Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/S
Lcom/google/android/smart/g; a (Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/String; Ljava/lang/S
```

Searching !

Strings

It is possible to search where a string is used:

```
In [11]: z = dx.tainted_variables.get_string(".apk")

In [12]: z
Out[12]: <androguard.core.analysis.analysis.TaintedVariable instance at 0xcabcd8c>

In [13]: z.show_paths(d)
R Lcom/google/android/smart/t; a ()V a-BB@0x3e 50
```

Fields

Like with strings, it is the same with a field, you can know where a field is read/write:

```
In [17]: z = dx.tainted_variables.get_field("Lcom/google/android/smart/s;", "a", "Lcom/google/android/smart/x;")

In [18]: z
Out[18]: <androguard.core.analysis.analysis.TaintedVariable instance at 0xbe7028c>

In [19]: z.show_paths(d)
R Lcom/google/android/smart/a; a ()V a-BB@0x0 8
R Lcom/google/android/smart/b; a ()V a-BB@0x0 28
R Lcom/google/android/smart/b; a ()V a-BB@0x52 6a
R Lcom/google/android/smart/b; a ()V a-BB@0xb0 b8
R Lcom/google/android/smart/b; a ()V a-BB@0xe0 e8
R Lcom/google/android/smart/b; a ()V a-BB@0xfc 104
R Lcom/google/android/smart/b; a ()V a-BB@0x10e 116
```

```
[...]
```

Packages

You can search for a specific package by using a regexp:

```
In [22]: show_Path(d, dx.tainted_packages.search_packages("Landroid/telephony/"))
Lcom/google/android/smart/g; a ()V (@a-BB@0x3c-0x44)  ---> Landroid/telephony/gsm/GsmCellLocation; getCid ()I
Lcom/google/android/smart/g; a ()V (@a-BB@0x3c-0x62)  ---> Landroid/telephony/gsm/GsmCellLocation; getLac ()I
[...]
```

or you can used prebuilt method:

- crypto: search_crypto_packages
- telephony: search_telephony_packages
- net: search_net_packages

```
In [28]: show_Path(d, dx.tainted_packages.search_crypto_packages())
Lcom/google/android/smart/s; a ()Ljava/lang/String; (@a-BB@0x3c-0x9a)  ---> Ljavax/crypto/SecretKey; getEncoded ()[B
Lcom/google/android/smart/s; a ()Ljava/lang/String; (@a-BB@0x3c-0xb4)  ---> Ljavax/crypto/Cipher; getInstance (Ljava/la
Lcom/google/android/smart/s; a ()Ljava/lang/String; (@a-BB@0x3c-0xbe)  ---> Ljavax/crypto/Cipher; init (I Ljava/securit
Lcom/google/android/smart/s; a ()Ljava/lang/String; (@a-BB@0x3c-0xc4)  ---> Ljavax/crypto/Cipher; doFinal ([B)[B
[...]
```

Methods

You can search for a specific method by using the search_methods function, with regexp arguments:

```
In [14]: dx.tainted_packages.search_methods?
Docstring:
@param class_name : a regexp for the class name of the method (the package)
@param name : a regexp for the name of the method
@param descriptor : a regexp for the descriptor of the method

@rtype : a list of called methods' paths


In [12]: show_Paths(d, dx.tainted_packages.search_methods(".", "getDeviceId", "."))
1 Lsergio/samples/searchingtest/MainActivity;->onCreate(Landroid/os/Bundle;)V (0x20) ---> Landroid/telephony/TelephonyM
```

Objects

Dynamic Code Loading

It's interesting to search automatically if an application use DexClassLoader in order to load dynamically dex files. For that, you can use the *show_Dyncode* function with the analyzed dex:

```
In [41]: a, d, dx = AnalyzeAPK("./apks/malwares/anserverbot/06457902965e95183211fa5e36aa8b6d860ba51891d666fcccaf52810db
In [42]: show_DynCode(dx)
Lcom/sec/android/providers/drm/Style; a (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; b (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; c (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; a (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; a (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; b (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; b (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; c (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
Lcom/sec/android/providers/drm/Style; c (Ljava/io/File; Ljava/lang/String; Ljava/lang/String; [Ljava/lang/Object;)Ljava
In [13]: is_dyn_code(dx)
Out[13]: True
```

Native Code

```
is_native_code(dx)
show_NativeCode(dx)
```

Reflection

```
is_reflection_code(dx)
show_ReflectionCode(dx)
```

XREF

You can create and export the XREF directly if you have associated the DalvikVMFormat object and the VMAnalysis object:

```
d.set_vmanalysis( dx )
```

After that you can create (and export in python namespace) the XREF:

```
d.create_xref()
```

You will have two objects (XREF class) per method: XREFfrom and XREFto. The first object represents where this method is called, and the second one represents which method is called:

```
In [30]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom
Out[30]: <androguard.core.bytecodes.dvm.XREF instance at 0xd0b774c>

In [31]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFto
Out[31]: <androguard.core.bytecodes.dvm.XREF instance at 0xd0b778c>

In [38]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.Lcom_google_android_smart_f__a__V
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.Lcom_google_android_smart_k__a__V
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.Lcom_google_android_smart_l__a__V
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.Lcom_google_android_smart_u__a__V
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.__class__
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.__doc__
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.__init__
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.__module__
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.add
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.items

In [38]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFto.
d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFto.__class__   d.CLASS_Lcom_google_android_smart_s.I
[...]


In [33]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.XREFfrom.items
Out[33]:
[(<androguard.core.bytecodes.dvm.EncodedMethod instance at 0xb20ba8c>,
    [<androguard.core.analysis.analysis.PathP instance at 0xcaf774c>]),
 (<androguard.core.bytecodes.dvm.EncodedMethod instance at 0xb1fed2c>,
    [<androguard.core.analysis.analysis.PathP instance at 0xc93ac4c>,
     <androguard.core.analysis.analysis.PathP instance at 0xc93aeec>,
     <androguard.core.analysis.analysis.PathP instance at 0xc93f3ec>,
     <androguard.core.analysis.analysis.PathP instance at 0xc93f68c>,
     <androguard.core.analysis.analysis.PathP instance at 0xc9447ac>,
     <androguard.core.analysis.analysis.PathP instance at 0xc944a4c>,
     <androguard.core.analysis.analysis.PathP instance at 0xc9487ac>,
     <androguard.core.analysis.analysis.PathP instance at 0xc948a4c>,
     <androguard.core.analysis.analysis.PathP instance at 0xc94c7ac>,
     <androguard.core.analysis.analysis.PathP instance at 0xc94ca4c>]),
  (<androguard.core.bytecodes.dvm.EncodedMethod instance at 0xb20284c>,
    [<androguard.core.analysis.analysis.PathP instance at 0xc9bcdec>]),
  (<androguard.core.bytecodes.dvm.EncodedMethod instance at 0xb2029cc>,
    [<androguard.core.analysis.analysis.PathP instance at 0xc9cc90c>,
     <androguard.core.analysis.analysis.PathP instance at 0xc9ccdcc>,
     <androguard.core.analysis.analysis.PathP instance at 0xc9ce22c>])]
```

As you can see you can use the completion to have a direct access to the corresponding method. And if you display a method, you will have at the end the same information:

```
In [34]: d.CLASS_Lcom_google_android_smart_s.METHOD_a_Ljava_lang_StringV.pretty_show()
        METHOD access_flags=17 (Lcom/google/android/smart/s; a,(Ljava/lang/String;)V)
[...]
********************************************************************************
F: Lcom/google/android/smart/u; a ()V ['6a']
F: Lcom/google/android/smart/f; a ()V ['1de', '1ec', '2da', '2e8', '426', '434', '4f2', '500', '5be', '5cc']
F: Lcom/google/android/smart/k; a ()V ['1a6']
F: Lcom/google/android/smart/l; a ()V ['18a', '1aa', '1ca']
```

DREF

The DREF is used to know where a specific field is used. Like XREF you need to have an association between DalvikVMFormat and VMAnalysis objects and after you can create DREF:

```
d.create_dref( dx )
```

and each field will have two objects (DREF class) per field: DREFr and DREFw. The first object represents where a field is read and the second one represents where a field is written:

```
In [35]: d.CLASS_Lcom_google_android_smart_s.FIELD_a.DREFr
Out[35]: <androguard.core.bytecodes.dvm.DREF instance at 0xd1eb96c>

In [36]: d.CLASS_Lcom_google_android_smart_s.FIELD_a.DREFw
Out[36]: <androguard.core.bytecodes.dvm.DREF instance at 0xd1eb98c>

In [37]: d.CLASS_Lcom_google_android_smart_s.FIELD_a.DREFw.
d.CLASS_Lcom_google_android_smart_s.FIELD_a.DREFw.Lcom_google_android_smart_s__init__Landroid_content_ContextV
[...]

In [37]: d.CLASS_Lcom_google_android_smart_s.FIELD_a.DREFw.items
Out[37]:
[(<androguard.core.bytecodes.dvm.EncodedMethod instance at 0xb20b04c>,
  [<androguard.core.analysis.analysis.Path instance at 0xca7fd2c>])]
```

Reflection

Exceptions

Changing the start offset of analysis

We used a Linear Travel algorithm, but I hope that for the 2.0 release we will have a recursive algorithm to deal with such techniques. But, if you see such techniques in real samples, you can use (but be carefull, it will not help you in all cases !) the set_code_idx method, it will help to change the offset where the disassemble of the buffer starts.

```
In [5]: d.CLASS_Lorg_dexlabs_poc_dexdropper_DropActivity.METHOD_init.pretty_show()
########## Method Information
Lorg/dexlabs/poc/dexdropper/DropActivity;-><init>()V [access_flags=public constructor]
########## Params
local registers: v0...v1
- return:void
####################
******************************************************************************
<init>-BB@0x0 :
        0  (00000000) if-eq               v0, v0, +9 [ <init>-BB@0x4 <init>-BB@0x4 ]

<init>-BB@0x4 :
        1  (00000004) fill-array-data     v0, +3 (0xa)
        2  (0000000a) fill-array-data-payload 'p\x10\x08\x00\x01\x00\x12P#\x00\x00\x02&\x00\x07\x00\x00\x0
x00\x01\x00\x12\x50\x23\x00\x00\x02\x26\x00\x07\x00\x00\x00\x5b\x10\xcf\x02\x0e\x00\x00\x00\x00\x03\x01\x0

******************************************************************************
########## XREF
####################

In [6]: d.CLASS_Lorg_dexlabs_poc_dexdropper_DropActivity.METHOD_init.set_code_idx(0x12)

In [7]: d.CLASS_Lorg_dexlabs_poc_dexdropper_DropActivity.METHOD_init.pretty_show()
########## Method Information
Lorg/dexlabs/poc/dexdropper/DropActivity;-><init>()V [access_flags=public constructor]
########## Params
local registers: v0...v1
- return:void
####################
******************************************************************************
<init>-BB@0x0 :
        0  (00000000) invoke-direct       v1, Landroid/app/Activity;-><init>()V
        1  (00000006) const/4             v0, #+5
        2  (00000008) new-array           v0, v0, [B
        3  (0000000c) fill-array-data     v0, +7 (0x1a)
        4  (00000012) iput-object         v0, v1, Lorg/dexlabs/poc/dexdropper/DropActivity;->exit [B
        5  (00000016) return-void

<init>-BB@0x18 :
        6  (00000018) nop
        7  (0000001a) fill-array-data-payload 'sjbdw\x00' | \x73\x6a\x62\x64\x77\x00

******************************************************************************
########## XREF
```

Session

Save/Load

You can save and restore (with cPickle) an analysis session of an Android App, by using save_session and load_session functions in the shell.

```
In [1]: a, d, dx = AnalyzeAPK("./apks/porn
./apks/pornoplayer.apk      ./apks/pornoplayer.apk.txt  ./apks/pornoplayer2.apk

In [1]: a, d, dx = AnalyzeAPK("./apks/pornoplayer.apk
./apks/pornoplayer.apk      ./apks/pornoplayer.apk.txt

In [1]: a, d, dx = AnalyzeAPK("./apks/pornoplayer.apk")

In [2]: a, d, dx
Out[2]:
(<androguard.core.bytecodes.apk.APK instance at 0x35aa6c8>,
 <androguard.core.bytecodes.dvm.DalvikVMFormat at 0x36bda90>,
 <androguard.core.analysis.analysis.uVMAnalysis instance at 0x36db878>)

In [3]: save_session([a, d, dx], "w00t.ag")

In [4]:
Do you really want to exit ([y]/n)? y

desnos@t0t0:~/androguard$ ./androlyze.py -s
Androlyze version 1.5
In [1]: a, d, dx = load_session("w00t.ag")

In [2]: a, d, dx
Out[2]:
(<androguard.core.bytecodes.apk.APK instance at 0x22ef638>,
 <androguard.core.bytecodes.dvm.DalvikVMFormat at 0x23af190>,
 <androguard.core.analysis.analysis.uVMAnalysis instance at 0x2472638>)
```

Annotation

You can annotate an instruction or a method, and it will be display on the screen (add_inode, add_node functions).

```
In [6]: d.CLASS_Lorg_media_player_MoviePlayer.METHOD_init.add_note("Constructor of an evil class !")

In [7]: d.CLASS_Lorg_media_player_MoviePlayer.METHOD_init.pretty_show()
########## Method Information
Lorg/media/player/MoviePlayer;-><init>()V [access_flags=public constructor]
########## Notes
# Constructor of an evil class !
####################
########## Params
local registers: v0...v0
- return:void
####################
**************************************************************************
<init>-BB@0x0 :
        0  (00000000) invoke-direct        v0, Landroid/app/Activity;-><init>()V
        1  (00000006) return-void

**************************************************************************
```

```
In [4]: d.CLASS_Lorg_media_player_MoviePlayer.METHOD_onCreate.add_inote("Send a SMS to a premium number", 21)

onCreate-BB@0x48 :
       17 (00000048) const-string         v1, '4161'
       18 (0000004c) const-string         v3, 'dx427123'
       19 (00000050) move-object          v4, v2
       20 (00000052) move-object          v5, v2
       # Send a SMS to a premium number
       21 (00000054) invoke-virtual/range v0 ... v5, Landroid/telephony/SmsManager;->s
d/app/PendingIntent;)V
       22 (0000005a) add-int/lit8         v7, v7, #+1
       23 (0000005e) goto                 -e [ onCreate-BB@0x42 ]
```

Bypass non ascii characters in names

Renaming names

It is possible to change the name of classes, methods or fields when you have obfuscation (ie: with proguard).

It is possible to rename a class, a method or a field, by using the set_name method in each corresponding object, and it is really interesting when a sample has been obfuscated.

```
In [3]: d.CLASS_Lorg_media_player_MoviePlayer.set_name("LEvil;")

In [4]: d.CL
d.CLASS_LEvil                                          d.CLASS_Lorg_me_androi
d.CLASS_Lorg_me_androidapplication1_DataHelper         d.CLASS_Lorg_me_androi
d.CLASS_Lorg_me_androidapplication1_DataHelper_OpenHelper  d.CLASS_Lorg_me_androi
d.CLASS_Lorg_me_androidapplication1_MoviePlayer        d.CLASS_Lorg_media_play
d.CLASS_Lorg_me_androidapplication1_R                  d.CLASS_Lorg_media_play
d.CLASS_Lorg_me_androidapplication1_R_attr             d.CLASS_Lorg_media_play

In [4]: d.CLASS_LEvil.METHOD_init.pretty_show()
########## Method Information
LEvil;-><init>()V [access_flags=public constructor]
########## Params
local registers: v0...v0
- return:void
####################
********************************************************************************
<init>-BB@0x0 :
        0  (00000000) invoke-direct        v0, Landroid/app/Activity;-><init>()V
        1  (00000006) return-void

********************************************************************************
```

```
In [9]: d.CLASS_Lorg_media_player_DataHelper.METHOD_was.pretty_show()
########## Method Information
Lorg/media/player/DataHelper;->was()V [access_flags=public]
########## Params
local registers: v0...v1
- return:void
####################
********************************************************************************
was-BB@0x0 :
        0  (00000000) iget-object          v0, v1, Lorg/media/player/DataHelper;->insertStmt Landroi
        1  (00000004) invoke-virtual       v0, Landroid/database/sqlite/SQLiteStatement;->executeIns
        2  (0000000a) return-void

********************************************************************************
########## XREF
F: LEvil; onCreate (Landroid/os/Bundle;)V 60
####################
```

```
In [4]: d.CLASS_Lorg_media_player_DataHelper.METHOD_was.set_name("check_field_db")

In [5]: d.CLASS_Lorg_media_player_DataHelper.METHOD_
d.CLASS_Lorg_media_player_DataHelper.METHOD_canwe          d.CLASS_Lorg_media_player_DataHelper.METHOD_check_field_db

In [5]: d.CLASS_Lorg_media_player_DataHelper.METHOD_check_field_db.pretty_show()
########## Method Information
Lorg/media/player/DataHelper;->check_field_db()V [access_flags=public]
########## Params
local registers: v0...v1
- return:void
####################
********************************************************************************
check_field_db-BB@0x0 :
        0  (00000000) iget-object          v0, v1, Lorg/media/player/DataHelper;->insertStmt Landroid/database/sqlite/S
        1  (00000004) invoke-virtual       v0, Landroid/database/sqlite/SQLiteStatement;->executeInsert()J
        2  (0000000a) return-void
```

Decompiler

Analyse APK/DEX/ODEX/AXML/ARSC with the API

Maybe you want to do your own script to analyse android applications, so I will show you how to do that.

Open the app

It depends of what kind of app you would like to analyse. If you have a classical APK file, the first thing to do is to import the apk module:

```
from androguard.core.bytecodes import apk
```

and to open your file:

```
a = apk.APK("pathtoyouruberfile.apk")
```

but maybe you don't have a path to your file system, and only a raw buffer:

```
a = apk.APK(rawbuffer, raw=True)
```

You have different options with the APK class but maybe you have already seen error with the zipmodule of python. You can try to load our fix modules by using the zipmodule option:

```
a = apk.APK(rawbuffer, raw=True, zipmodule=2)
```

But if your file is a DEX or ODEX format, you can do:

```
from androguard.core.bytecodes import dvm

d = dvm.DalvikVMFormat(rawbuffer)
d = dvm.DalvikVMFormat(open("pathtoyourfile.dex", "r").read())
d = dvm.DalvikOdexVMFormat(open("pathtoyourfile.odex", "r").read())

# you have only the apk and want to analyse the dex file in the apk
a = apk.APK(rawbuffer, raw=True)
d = dvm.DalvikVMFormat(a.get_dex())
```

AXML/ARSC

Get all classes

When you have a DalvikVMFormat object, you can access to all classes (ClassDefItem object):

```
d = dvm.DalvikVMFormat(a.get_dex())

for current_class in d.get_classes():
  print current_class
```

Of course you can get all methods/fields in the current class by using get_methods/get_fields functions, check the API.

Get all methods

You can access to all methods from all classes directly:

```
d = dvm.DalvikVMFormat(a.get_dex())

for current_method in d.get_methods():
  print current_method
```

each current method is an EncodedMethod object, and you have plenty of functions to get information(the class name, the name, the descriptor, the code ...)

Get all fields

```
d = dvm.DalvikVMFormat(a.get_dex())

for current_field in d.get_fields():
  print current_field
```

You will have an EncodedField object !

Instructions

I think that you are interesting to get instructions from a method. Each instruction is a Instruction object, depending of the format of the instruction, it can be a different class. But the main object will be Instruction.

Moreover, each instruction respect the definition of the dex format, and each instruction will have exactly the same arguments.

You have the old way, by accessing each object (EncodedMethod -> DalvikCode -> DCode)

```
for method in a.get_methods() :
    print method.get_class_name(), method.get_name(), method.get_descriptor()
    code = method.get_code()
    bc = code.get_bc()

    idx = 0
    for i in bc.get_instructions() :
        print "\t", "%x" % idx, i.get_name(), i.get_output()
        idx += i.get_length()
```

or a simple way by using the get_instructions of the EncodedMethod:

```
for method in a.get_methods():
    print method.get_class_name(), method.get_name(), method.get_descriptor()
    idx = 0
    for i in method.get_instructions():
        print "\t", "%x" % idx, i.get_name(), i.get_output()
        idx += i.get_length()
```

Search

CFG

You can build yourself the CFG if you wish, because you will have access to instructions. But if you are lazy, we have already did it for you.

You have a DalvikVMFormat object, and you must analyse it with a VMAnalysis class:

```
d = dvm.DalvikVMFormat(open(TEST, "r").read())
x = analysis.VMAnalysis(d)
```

Ok that's all, and now you can access to each basic blocks of a method, and next and previous blocks:

```
for method in d.get_methods():
    g = x.get_method(method)

    if method.get_code() == None:
      continue

    print method.get_class_name(), method.get_name(), method.get_descriptor()

    idx = 0
    for i in g.get_basic_blocks().get():
        print "\t %s %x %x" % (i.name, i.start, i.end), '[ NEXT = ', ', '.join( "%x-%x-%s" % (j[0], j[1], j[2].get_name

        for ins in i.get_instructions():
            print "\t\t %x" % idx, ins.get_name(), ins.get_output()
            idx += ins.get_length()

        print ""
```

Source Code

Now, you want the source code dude ? haha ok :)

The first things is to setup a decompiler, but we advised to use DAD which is our internal decompiler. You must import the correct module

```
from androguard.decompiler.dad import decompile
```

and now when you access to each method, you can decompile it by using a DvMethod object, and the get_source method:

```
vm = dvm.DalvikVMFormat(open("myfile.dex", "r").read())
vmx = analysis.VMAnalysis(vm)

# CFG
for method in vm.get_methods():
    mx = vmx.get_method(method)

    if method.get_code() == None:
      continue

    print method.get_class_name(), method.get_name(), method.get_descriptor()

    ms = decompile.DvMethod(mx)
    # process to the decompilation
    ms.process()

    # get the source !
    print ms.get_source()
```

Automatic Analysis

It is possible to analyse multiple android apps (APK, DEX, ODEX) by using different threads.

For that we have a specific module AndroAuto, the main usage is the following:

```
from androguard.core.analysis import auto
```

```
# create the new analyser
aa = auto.AndroAuto(settings)
# run the analysis
aa.go()
# dump the result
aa.dump()
```

where the variable settings is a dict:

```
class AndroLog:
  def __init__(self, id_file, filename):
    self.id_file = id_file

settings = {
 "my": auto.DirectoryAndroAnalysis(options.directory),
 "log": AndroLog,
 "max_fetcher": 3,
}
```

which specify an object to use to analyse all apps, and a log class which will be use for each app to do what you want :), and finaly the number of threads that you would like.

The first object must have specific methods, and can be inherit of the the DefaultAndroAnalysis class.

So it is possible to filter quickly which kind of android apps you would like to analyse (APK/DEX/ODEX). And for each step in the algorithm, you can stop or continue (analyse the APK, the classes.dex ...)

Graphical Export

DOT

PNG

GEXF

GML

Sublime Text 2 Plugin

With this plugin you can open APK/DEX/ODEX/AXML/ARSC files directly in the editor.

Installation (Windows/Linux/OSX)

You must copy the archive and extract it in the Packages directory for Sublime Text.

Key Binding

By default the key binding are the following (you can change them if you wish):

- ctfl+f5: open a APK/DEX/ODEX/AXML/ARSC files
- f5: switch to Dalvik Bytecodes to source codes, switch to AndroidManifest.xml summary to original xml file
- ctrl+f6: get callers methods for a method, or read access for a field
- ctrl+f7: get callees methods for a method, or written access for a field
- ctrl+f8: reset the plugin

Mouse Binding

On an APK file:

- double click on a filename to see the content.

On a Dex file:

- double click on a class/method/field to see the content.

Similarities/Differences

Please read this tutorial

Is it a malware ?

If you would like to check if a sample is present in the Androguard malware/adware database or in your own database, you can do the following things:

```
In [1]: from elsim.elsign import dalvik_elsign
In [44]: m = dalvik_elsign( "signatures/dbandroguard", "signatures/dbconfig", False, ps = dalvik_elsign.PublicSignature

In [46]: m.check_dex_direct(d, dx)
Out[46]: (None, [])

n [47]: a1, d1, dx1 = AnalyzeAPK("./apks/malwares/DroidDream/Magic Hypnotic Spiral.apk")

In [48]: m.check_dex_direct(d1, dx1)
Out[48]: (u'DroidDream', [[69, 0.16257669031620026]])
```

The first thing is to create a new *MSignature* object by giving the filename of the database and the configuration. And after to call the *load* method. And you can use one of the available methods to check if a signature is detected:

```
In [49]: m.ch
m.check_apk         m.check_dex         m.check_dex_direct
```

Comment by jtag.o...@gmail.com, Aug 19, 2012

Notes: **the correct call for extending is `d.create_python_export()`** the pretty print setup command is not working, maybe `init_print_colors()`?

► Sign in to add a comment

Terms - Privacy - Project Hosting Help

Powered by Google Project Hosting