FileScan user manual

script version: 1.2.1

Giovanni Pessiva
giovanni.pessiva@gmail.com

Contents

1 The FileScan project

FileScan has been realised by Giovanni Pessiva as a part of the work for his Master's degree in Computer Engineering at the Polytechnic University of Turin. The project was developed in collaboration with the Telecom Italia researchers Rosalia D'Alessandro and Madalina Baltatu, and under the supervision of the TORSEC security group.

2 Installation

2.1 Python dependencies

FileScan requires many libraries, some of which are included in the Python Standard Library; therefore, those libraries should be available in the default installation of Python. Namely, they are: *sys*, *os*, *shutil*, *StringIO*, *cStringIO*, *re*, *hashlib*, *subprocess*, *shlex*, *zipfile*, *time*, *gzip*, *tarfile*.

Other libraries must be installed, since they are unofficial modules:

- *magic*: https://github.com/ahupp/python-magic
- *rarfile*: http://developer.berlios.de/projects/rarfile
- *fuzzy*: http://pyfuzzy.sourceforge.net/

Also, the framework **Androguard** is required, and in particular its modules *core.androconf* and *core.bytecodes.apk*. **Androguard** can be downloaded from its official page: http://code.google.com/p/androguard/; the minimum version required is 1.9, since it introduced the resource management (*androarsc*).

2.2 Magic database file

To avoid any problem in its categorisation method, FileScan is designed to look for a custom magic database file (*magic.mgc*) in its path; if such file is not found, the system file will be used. This was done to improve as much as possible the portability, but generally the custom file should not be required; therefore, you can ignore this dependency. The only features that (potentially) could be influenced by a totally different magic file are: *dex* code detection, *arsc* file detection, non-critical files categorisation. The custom *magic.mgc* file is distributed along with FileScan.

2.3 Unofficial Python patch

During the analysis, an exception could be raised; when the Python *zipfile* module tries to decompress a particular set of *apk* files, it fails and returns the exception: "*unpack requires*

a string argument of length 4". This is caused by an issue that occurs with *zip* files that violate the *zip* format specifications; namely, they declare a length of 0 for the extra part, but have a valid string or some padding (an arbitrary amount of null characters \x00) in that field (more information here: http://bugs.python.org/issue14315). While many other archival manager tools simply ignore this issue, the Python *zip* module has a strict approach and raises that exception; this results in being unable to analyse these *apk* files. A non-official patch has been proposed, but the change has not been implemented in the official release yet; the patch can be found here: http://bugs.python.org/file24902/fix_zipfile_extra.patch.

3 API

Most of the methods of FileScan are intended to be used by other Python scripts, so they return Python objects (lists or dictionaries) instead of printing the results. They can be divided into five categories, depending on how their output is intended to be used: in order to provide general information about the application, to identify URL addressed/phone numbers/shell scripts, to identify suspect files, to identify infected files, or to compute the risk scores. In order to use the first category of APIs, it is just necessary to create an instance of the FileScan object, for example:

```
1 file_scan = FileScan(''/home/giova/samples/application.apk'')
```

As for the other methods, first they require the *apk* to be analysed; this can be done by invoking the method `analyze_files()`, which will perform a full and recursive analysis on the FileScan object, returning a list of risk values.

```
1 risk_values = file_scan.analyze_files()
```

After this method has been used, the other methods will be able to provide the information detected during the analysis; the time required by the method `analyze_files()` can be retrieved using the method `get_analysis_time()`. A method can also be used to print a complete report of the analysis: `report()`; the output can also be printed to a file, passing its handle as a parameter.

3.1 General information

A very general list of the file types found in the analysed *apk* is returned by `get_types_count()`; it gives as output a dictionary that associates a file type to its occurrences in the package. The purpose of this method is to provide a general overview on the package, identifying as many files as possible. The types are the following:

- `multimedia`: images, video, audio files;
- `android xml`: binary XML files;
- `android apk`: Android applications;
- `compressed archive`: zip, gzip, rar, tar, etc.;
- `xml`: XML files in plain text;
- `text`: textual files;

- `dex`: *dex* code files;
- `elf`: native executables or shared libraries;
- `android resource`: *resource.arsc* files;
- `other`: other common (but not interesting) files;
- `empty file`: files with size = 0 KB;
- `unknown binary`: binary files with unknown format.

The methods can be used as follows:

```
for k,v in file_scan.get_types_count().iteritems() :
        print ''Type:'', k, '',occurrences:'', v
```

A simple list of every file in the application, including the files inside compressed archives, is returned by the method `get_files_list()`; for a list which include only the *apk* files embedded inside the application, the method `get_apk()` can be used.

The SHA1 hashcode of the *apk* file can be obtained with the method `get_sha1()`.

Dictionaries that associate *dex* and ELF files to their hash codes can be generated by the methods `get_dex_checksum()` and `get_elf_checksum()`. They require as input the function to use; acceptable values are: "md5", "sha1", "sha224", "sha256", "sha384", "sha512".

3.2 URLs, phone numbers, shell scripts

FileScan can generate lists of the following data, associated to a list of the files in which they have been found:

- URL addressed (along with their domain);
- Parametrised and escaped URL addressed (along with their domain);
- Critical shell commands;
- Potential phone numbers.

The related methods are: `get_urls()`, `get_urls_encoded()`, `get_scripts()`, `get_numbers()`. The following examples shows how these functions can be used:

```
urls, domains, urls_files = file_scan.get_urls()
i = 0
while i<len(urls) :
        print ''URL in file'', urls_files[i], ''>'', urls[i]
        i+=1

urls, domains, urls_files = file_scan.get_urls_encoded()
i = 0
while i<len(urls) :
        print "Encoded URL in file'', urls_files[i], ''>'', urls[i]
        i+=1
```

```
12
13 numbers, numbers_files = file_scan.get_numbers()
14 i = 0
15 while i<len(numbers) :
16         print ''Potential phone number in file'', numbers_files[i],
               ''>'', numbers[i]
17         i+=1
18
19 scripts, scripts_files = file_scan.get_scripts()
20 i = 0
21 while i<len(scripts) :
22         print ''Shell command in file'', scripts_files[i], ''>'',
               scripts[i]
23         i+=1
```

3.3 Suspect files

FileScan is able to identify textual files, *apk* files and ELF files that, in its opinion, have been renamed with unusual extensions, maybe with the intention of hiding them. A dictionary of those files (which associates to each file the description of its real type) can be retrieved from the method `get_suspect_files()`.

3.4 Infected files

FileScan is able to check the *dex* and ELF files, comparing their checksums to known malwares. In order to do so, first of all it uses an internal list of known ELF exploits, which have been found in different samples of malware applications. A dictionary that associates the files to the detected exploit is returned by the method `get_elf_infected()`.

Also, FileScan is able to perform queries to the Hash Malware Registry (http://www.team-cymru.org/Services/MHR/), to retrieve the detection rate for *dex* files and ELF files. The returned information is accessible through the methods `get_dex_detection_rate()` for the *dex* code and `get_elf_detection_rate()` for the native executables; they both return a dictionary that associates to each file its percentage detection rate.

3.5 Risk scores

One of the main features of FileScan is the possibility to compute a final score that indicate the potential harm that the analysed application could pose to the user. This score is computed by combining other partial scores, which are returned by the method `analyze_files()`. For example, they can be visualised with the following snippet of code:

```
1 risk_values = file_scan.analyze_files()
2 print ''InfectedDex ='', risk_values[0]
3 print ''InfectedElf ='', risk_values[1]
4 print ''HiddenApk ='', risk_values[2]
5 print ''HiddenElf ='', risk_values[3]
6 print ''HiddenText ='', risk_values[4]
7 print ''EmbeddedApk ='', risk_values[5]
```

```
print ''Shell ='', risk_values[6]
print ''ShellInstall ='', risk_values[7]
print ''ShellPrivilege ='', risk_values[8]
print ''ShellOther ='', risk_values[9]
```

The final and global risk score is returned by the method `get_risk_score()`.

It could be useful to print these values in the CSV (comma-separated values) format; this can be done with the following piece of code:

```
risk_values = file_scan.analyze_files()
string = ''\''''+os.path.basename(apk)+''\''''
string += ''';''''+file_scan.get_sha1()
string += ''';''''+str(file_scan.get_risk_score())
string += ''';''''+str(risk_values[0])
string += ''';''''+str(risk_values[1])
string += ''';''''+str(risk_values[2])
string += ''';''''+str(risk_values[3])
string += ''';''''+str(risk_values[4])
string += ''';''''+str(risk_values[5])
string += ''';''''+str(risk_values[6])
string += ''';''''+str(risk_values[7])
string += ''';''''+str(risk_values[8])
string += ''';''''+str(risk_values[9])
print string
```

4 Example

The following example is a complete script, able to accept as input an *apk* file or a directory, to perform an analysis and to print the analysis report.

```
#!/usr/bin/env python

import sys, os
from time import time
from optparse import OptionParser
from filescan import *

option_0 = { 'name' : ('-i', '--input'), 'help' : 'file : use this
   filename', 'nargs' : 1 }
option_1 = { 'name' : ('-d', '--directory'), 'help' : 'directory :
   examine all apks from this directory', 'nargs' : 1 }
options = [option_0, option_1]

def test(apk) :
    try :
        analysis_time = 0
        start = time()
        file_scan = FileScan(apk)
        risk_values = file_scan.analyze_files()
```

```
18          analysis_time = time() - start
19                  print ``\'''+os.path.basename(apk)+``\'' analysed
                        in ''+str(analysis_time)
20          file_scan.report()
21      except Exception, e:
22          print ``\''Exception raised (for apk:'',apk+``):''+str(e)
              +``\"''
23
24  def main(options, arguments) :
25      if options.input is None and options.directory is None :
26          print 'waiting for file(s) to analyze'
27          return
28      if options.input != None :
29          test(options.input)
30      else :
31          dirList=os.listdir(options.directory)
32          for fname in dirList:
33              if fname.lower().endswith(``.apk'') :
34                  test(options.directory + fname)
35
36  if __name__ == ``__main__'' :
37      parser = OptionParser()
38      for option in options :
39          param = option['name']
40          del option['name']
41          parser.add_option(*param, **option)
42
43      options, arguments = parser.parse_args()
44      sys.argv[:] = arguments
45      main(options, arguments)
```