FileScan developer manual

script version: 1.2.2

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 Script structure

3.1 Analysis flow

The analysis of Android applications is composed by three sequential steps:

1. A new instance of the FileScan class is created, giving as input parameter the path to the *apk* file;
2. The method `analyze_files()` is invoked on the FileScan object created in the previous step;
3. One or more methods are invoked in order to retrieve the results of the analysis.

The first step corresponds to the `__init__()` function, that is the constructor of the FileScan object. First of all, it decompresses all the files inside the package; since FileScan can be used recursively on the archives that could be found inside the *apk* file, it accepts not only the *zip* format, but also *tar*, *gzip* and *rar* files. Then it checks every file, using the method `_classify_file()`. For every archive file found during the analysis, the method `_scan_archive()` is invoked, which will instantiate a FileClass object therefore performing a recursive analysis.

The method `_classify_file()` is invoked for every file found inside the initial *apk* file; it tries to identify the file, comparing its MIME type against different cases, and stores some data about it. If it succeeds in finding the exact type of the file, and it happens to be an interesting format (text, *dex*, ELF, archive), the method will also save the raw data, in order to be able to analyse the file later (when the method `analyze_files()` is invoked). This part relies on the string returned by the *magic* module functions; therefore, unexpected values originated by the system magic database file could cause some problems. This is not very likely to happen; however, if available, a custom magic database file is used insted of the one of the system, in order to prevent any issue.

3.2 Analysis process

After the FileScan object has been initialised, it is possible to use the method `analyze_files()` to perform the real analysis process, which will produce a final risk score, along with other data. The `analyze_files()` method will check the application, considering also the files contained in embedded compressed archives, and decompiling the resource file *resource.arsc* in order to retrieve the *strings.xml* file (if the external **apktool** application is available). It will perform an analysis on:

- Textual files;

- ELF files;
- *dex* code files.

The textual files will be analysed with regular expressions, looking for URL addresses, URLs in parametric of escaped form, phone numbers, shell commands; the regex are saved as properties of the FileScan object (`re_url1`, `re_url2`, `re_sms1`, `re_script`).

For the URL addresses, the domain part is stored separately to the complete address. Also, addresses containing common domains (“*developer.android.com*”, “*schemas.android.com*”, “*w3.org*”, “*wc3.org*”, “*apple.com/DTD*”, “*apache.org*”, “*jquery.com*”, “*developers.facebook.com*”) are ignored.

Files which have extremely long lines will not be checked, since the time requested by the regular expressions will be unacceptably long; if a file contains single lines which are longer than 5000 characters it is not checked for shell commands, if longer than 10000 neither for URL addresses. Not restriction are posted for the phone number regular expression, since it is much more simpler.

ELF files are checked against the hashcodes of known exploits found in different samples of malware applications; those hashcodes are contained in a dictionary stored as property of the FileScan object (`known_infected_elf`), which associates the hash to the name of the exploit.

ELf and *dex* files are checked with the lookup service of the Malware Hash Registry, which is contacted in the method `get_detection_rate()`, using the private method `_query_mhr()`. This feature can be tested through the method `_is_mhr_reachable()`, which will return `True` if the online service is reachable, and `False` otherwise.

FileScan also checks the extensions of textual, ELF and *apk* files to see if they can be considered valid or they could have been changed in order to hide them. It takes this decision relying on a black list for textual files (“*png*”, “*jpg*”, “*jpeg*”, “*gif*”, “*bmp*”, “*wav*”, “*ogg*”, “*mp3*”, “*ttf*”, “*zip*”), and white lists for ELF (“*so*”, “*exe*”) and *apk* (“*apk*”, “*jar*”) files. If those files have unexpected extension (or even none, for ELF and *apk* files), they are considered to be suspect.

3.3 Fuzzy system

A fuzzy system is used to compute the final risk score.

In order to represent the information retrieved by FileScan, the following risk variables have been defined:

- *HiddenElf*: an ELF file is not in the standard directory and has an unexpected extension;
- *InfectedElf*: an ELF file is detected as infected;
- *EmbeddedApk*: the analysed *apk* contains another *apk*;
- *HiddenApk*: an embedded *apk* file has an unexpected extension;
- *InfectedDex*: a *dex* file is detected as infected;
- *HiddenText*: a textual file has an invalid extension;
- *Shell*: a shell script has been found;

- *ShellInstall*: a file has been found, which contains shell commands able to install programs;
- *ShellPrivilege*: a file has been found, which contains shell commands usable to perform a privilege escalation;
- *ShellOther*: a file has been found, which contains dangerous shell commands.

All those risk categories have an associated value of 1; their true meaning and importance is expressed in the fuzzy rules. New fuzzy rules can be added in the method `_create_system_risk()` of the class RiskIndicator.

4 FileScan class properties

Generally, the properties of FileScan are not associated to getter or setter functions; their value is set during the analysis, and can be retrieved through some of the APIs.

4.1 Variables

Some of the parameters passed to the FileScan contructor are saved as properties:

- `apk_export_path` (default:

    ): path where embedded apk files will be exported, if any will be found;

    `enable_net_connection` (default:

      True): if `False`, FileScan will not perform queries to the online service Malware Hash Registry;

      `ignore_cmd` (default:

        False): if `True`, the regular expressions for URL addresses will not be used;

        `ignore_url` (default:

          False): if `True`, the regular expressions for shell commands will not be used;

          `ignore_sms` (default:

            False): if `True`, the regular expression for phone numbers will not be used.

During the initialisation of the FileScan object, all the data useful for the analysis are stored as properties. Theses properties are the following:

- `files`: dictionary that stores the raw data of the interesting files;
- `files_types_descr`: dictionary that stores the description of the type of the files;
- `files_types_mime`: dictionary that stores the MIME type of the files ;
- `files_compressed_filescan`: dictionary that stores the FileScan object created for every compressed archive file;

- `types`: dictionary that associates each file type to the number of files with that type (the types are: "*multimedia*", "*android xml*", "*android apk*", "*compressed archive*", "*xml*", "*text*", "*dex*", "*elf*", "*android resource*", "*unknown binary*", "*other*", "*empty file*");
- `filelist_*`: these variables are used to store lists of file names of that type, in particular `filelist_other` is used for uninteresting file formats, and `filelist_unknown` is used for unexpected file formats;
- `urls`: list of URL addresses found during the analysis;
- `urls_domains`: list of domains of the URLs in `urls`;
- `urls_files`: list of names of file where the URLs in `urls` have been found;
- `urls2`: list of escaped of parametric URL addresses found during the analysis;
- `urls2_domains`: list of domains of the URLs in `urls2`;
- `urls2_files`: list of names of file where the URLs in `urls2` have been found;
- `numbers`: list of possible phone numbers found during the analysis;
- `numbers_files`: list of names of file where the numbers in `numbers` have been found;
- `scripts`: list of shell commands found during the analysis;
- `scripts_files`: list of names of file where the shell commands in `scripts` have been found;
- `analysis_time`: time required by the method `analyze_files()`;
- `risks`: dictionary that associates risks to their value (the risks, defined as global variables, are `INFECTED_DEX_RISK`, `INFECTED_ELF_RISK`, `HIDDEN_APK_RISK`, `HIDDEN_ELF_RISK`, `HIDDEN_TXT_RISK`, `EMBEDDED_APK`, `SHELL_RISK`, `SHELL_INSTALL_RISK`, `SHELL_PRIVILEGE_RISK`, `SHELL_OTHER_RISK`);
- `script_count`: dictionary that associates each `SHELL_*` risk to its occurrences, counting each identical match once;
- `file_data`: raw data of the analysed application;
- `checksum_*`: these variables are used to store the hashcodes of the application, computed with MD5, SHA1 and SHA256.

4.2 Constants

There are properties which have fixed values:

- `common_extensions`: black list of extensions which textual files should not have;
- `elf_extensions`: white list of extensions which ELF files coud have;
- `apk_extensions`: white list of extensions which *apk* files coud have;
- `re_script`: dictionary of compiled regular expressions for identifying critical shell commands;
- `re_sms1`: compiled regular expression for identifying phone numbers;

- `re_url1`: compiled regular expression for identifying URL addresses;
- `re_url2`: compiled regular expression for identifying URL addresses with parameters or escaped characters;
- `re_url_known`: compiled regular expression for identifying common domains that should be ignored;
- `known_infected_elf`: dictionary that associates SHA1 hashcodes to known exploits.

5 FileScan class private methods

Every method in Python classes is public (can be accessed by functions from other classes), but the following are intended to be used only by other FileScan methods; they can be recognised by their name, for which this notation is used: "*_<method name>()*".

5.1 Initialisation methods

The `__init__` method uses some other private methods:

- `_classify_file()`: this function checks the MIME type and type description to categorise the files; it could be changed in order to make FileScan recognise more types of files, or perform different actions with each of them; also, this methods uses `_increment_type_count()` to update the files count for every file type;
- `_scan_archive()`: performs a recursive scan on a compressed archive found in the application, by instantiating a new FileScan object.

5.2 Analysis related methods

The method `analyze_files()`, which performs the main analysis process, uses the following private methods:

- `_analyze_file_readable()`: searches the file for URLs, commands, phone numbers;
- `_analyze_file_xml()`: searches the file for URLs and phone numbers;
- `_search_script()`: apply regular expressions to the input string, looking for shell commands;
- `_search_url()`: apply regular expressions to the input string, looking for URLs;
- `_search_sms()`: apply regular expressions to the input string, looking for phone numbers.

There are other private methods which corresponds to public methods, and share with them similar names; the difference is that they do not analyse any embedded archive, while the public methods are fully recursive on those archives. The methods are:

- `_suspect_files()`, used by `get_suspect_files()`;
- `_elf_checksum()`, used by `get_elf_checksum()`;
- `_dex_checksum()`, used by `get_dex_checksum()`.