Automation of analysis tasks with Volatility

In this laboratory you will practice with the development of automation processes for analysis tasks with Volatility. To do this, we will use the Python3 language to execute a *workflow* of analysis tasks. The objective of this laboratory is to describe the typical notions that you need to know in order to perform the integration of automatic analysis tasks in an optimal way.

From a software engineering point of view, remember that the software must be the more decoupled and modular the better, since it allows better reuse of the code and the ability to assess and improve system components independently. Below you will find various pieces of code that are responsible for performing specific tasks necessary to have a *workflow* of analysis tasks.

1 Reading configuration files

In order to automate analysis tasks with Volatility, there are two main elements to consider: the path to the Volatility application itself and, in case you are going to work with Python2, the path to the Python2 application. Reading these routes can be done through a configuration file, using the `configparse` Python3 package:

```
1  import configparser
2  import os
3
4  def load_cfg_file(config_file='config.ini'):
5      global VOL_BIN # global variable
6      global PYTHON2_BIN # global variable
7
8      config = configparser.ConfigParser()
9      config.read(config_file)
10     VOL_BIN = os.path.expanduser(
11                     os.path.normpath(config['DEFAULT']['VOL_BIN']))
12     PYTHON2_BIN = os.path.normpath(config['DEFAULT']['PYTHON2_BIN'])
```

The previous code allows us to read from a configuration file the routes of both applications. By default, the configuration file name is set to «`config.ini`». This file must have a format similar to the one shown in Listing 1, duly adapted to the paths of the applications in the particular system.

Listing 1: Content of the «`config.ini`» file to know the paths of Volatility and Python2 in the analysis machine.

```
[DEFAULT]
PYTHON2_BIN=/usr/bin/python2.7
VOL_BIN=~/volatility/vol.py
```

2 Create temporary working directory

Normally, during the execution of analysis tasks, a series of intermediate files are created that are not of interest for the analysis. It is best to have a temporary directory where these files are created, deleting it at the end of the execution of the program. To do this, the `tempfile` Python3 package can be used:

```
1 import tempfile
2 import os
3
4 def create_tmpdir():
5     tmp = os.path.join(tempfile.gettempdir(),
6                                '.{}'.format(hash(os.times())))
7     os.makedirs(tmp)
8     return tmp
```

This piece of code will return a path to a directory created with a different name on each run (the name depends on the current time) and located in the temporary working directory.

3 Volatility Execution

The following piece of code allows you to run a Volatility2 plugin on a dump given by parameter, also allowing you to specify the dump profile and any additional parameters that are needed. The first and second parameters are positional parameters, being the plugin name and the dump name respectively. All other parameters are keyword parameters.

This code executes the Python2.7 process along with Volatility2, and returns the output of the command and the execution code returned on success, or `None` and the execution code on failure.


```python
import subprocess
import os

def execute_vol_plugin(plugin_name, dump_file,
            plugin_folder='', profile='', extra_params='') -> (str, int):
    '''
    Execute a Volatility plugin and get the return (output + return code)
    '''
    # first build the command
    cmd = '{0} {1}'.format(PYTHON2_BIN, VOL_BIN)
    if plugin_folder != '':
        cmd += ' --plugins={0}'.format(plugin_folder)
    if profile != '': # append profile, if given
        cmd += ' --profile={0}'.format(profile)
    if ' ' in dump_file:
        print('[!] Please avoid white spaces in filenames, as in "{0}"'.
            format(dump_file))
    dump_file = os.path.abspath(os.path.expanduser(dump_file))
    # XXX Volatility not recognizes the input file when putting "{0}" ?
    cmd += ' -f {0} {1} {2}'.format(dump_file, extra_params, plugin_name)
    print(f'[*] Executing cmd: {cmd} ...', end='')
    _completed_process = subprocess.run(cmd.split(' '), capture_output=
        True, close_fds=True)
    print(' done!')
    # check error code
    if _completed_process.returncode != 0:
        print('Execution of command "{0}" finished with return code {1}'.
            format(cmd, _completed_process.returncode))
        print('[-] ERROR found: {0} '.format(_completed_process.stderr.
            decode("utf-8")))
        return None, _completed_process.returncode

    return _completed_process.stdout.decode("utf-8"), _completed_process.
        returncode
```

For instance, the following line of code allows you to run Volatility2 and the `pslist` plugin on the dump named «`alina1G.elf`»:

```python
output, returncode = execute_vol_plugin("pslist", "~/alina1G.elf",
                                         profile="Win7SP1x86")
```

If you want to get the output in JSON format (taking advantage of Volatility's unified output), it's as simple as adding the `extra_params` parameter to the above invocation:

```python
output, returncode = execute_vol_plugin("pslist", "~/alina1G.elf",
                                         profile="Win7SP1x86",
                                         extra_params="--output=json")
```

Finally, the output of Volatility can be transformed to a `DataFrame` of `pandas` in order to facilitate automatic processing. This can be achieved simply incorporating the following function into the code, which is responsible for processing a string (which will really be a dictionary) received as a parameter and transforming it into a `DataFrame`:


```python
import pandas as pd
import json

def JSONstr_to_DataFrame(output: str) -> pd.DataFrame:
    _json = json.loads(output)
    _list = [item for item in _json['rows']]
    # set rows
    df = pd.DataFrame.from_dict(_list)
    # set column names
    df.columns = _json['columns']
    return df
```

The advantage of having a `DataFrame` is that it allows you to perform typical column extraction operations very easily:

```
>>> print(df[['Name', 'PID']])
                Name   PID
0             System     4
1          smss.exe   268
2          csrss.exe   348
3       wininit.exe   384
4          csrss.exe   392
5      winlogon.exe   432
6      services.exe   476
7         lsass.exe   484
8           lsm.exe   492
9       svchost.exe   596
10   VBoxService.ex   660
11      svchost.exe   712
12      svchost.exe   764
13      svchost.exe   884
14      svchost.exe   928
15      audiodg.exe   988
16      svchost.exe  1096
17      svchost.exe  1228
18      spoolsv.exe  1308
19      svchost.exe  1344
20      svchost.exe  1448
21     taskhost.exe  1864
22          dwm.exe  1924
23     explorer.exe  1940
24     VBoxTray.exe   316
25   SearchIndexer.  1876
26   SearchProtocol   320
27   SearchFilterHo  1128
28   ALINA_CJLXYJ.e  1828
```


In the same way, you can also perform specific filters according to the value of a particular column:

```
print(df.loc[df['Name'].str.contains("ALINA", case=False)])
     Offset(V)            Name    PID   PPID   Thds   Hnds   Sess   Wow64
                                Start Exit
28   2245008456   ALINA_CJLXYJ.e   1828    628      2     47      1       0
    2019-09-21 12:07:04 UTC+0000
```

4 Summary

In summary, Python allows a relatively simple automation of analysis tasks since it has a multitude of packages and libraries that facilitate the work. For example, the `dlldump` plugin can be invoked to extract the DLLs of a given process (that meet certain requirements) and then parse them using the `ClamAV`[1] or perform a more detailed static analysis using `pefile`, `Capstone`, or similar tools. In Listing 2 you will find all the code used as an example throughout this workshop.

Listing 2: Analysis task automation sample code («`automated.py`» file).

```
import configparser
import os
import tempfile
import subprocess
import pandas as pd
import json

def load_cfg_file(config_file='config.ini'):
    global VOL_BIN # global variable
    global PYTHON2_BIN # global variable

    config = configparser.ConfigParser()
    config.read(config_file)
    VOL_BIN = os.path.expanduser(
                    os.path.normpath(config['DEFAULT']['VOL_BIN']))
    PYTHON2_BIN = os.path.normpath(config['DEFAULT']['PYTHON2_BIN'])

def create_tmpdir():
    tmp = os.path.join(tempfile.gettempdir(), '.{}'.format(hash(os.times
        ())))
    os.makedirs(tmp)
    return tmp

def execute_vol_plugin(plugin_name, dump_file, plugin_folder='', profile=
    '', extra_params='') -> (str, int):
    '''
    Execute a Volatility plugin and get the return (output + return code)
    '''
    # first build the command
    cmd = '{0} {1}'.format(PYTHON2_BIN, VOL_BIN)
    if plugin_folder != '':
```

[1] For instance, the `pyClamd` package can be useful for this purpose (available at `https://www.decalage.info/en/python/pyclamd`).


```
30          cmd += ' --plugins={0}'.format(plugin_folder)
31      if profile != '': # append profile, if given
32          cmd += ' --profile={0}'.format(profile)
33      if ' ' in dump_file:
34          print('[!] Please avoid white spaces in filenames, as in "{0}"'.
                format(dump_file))
35      dump_file = os.path.abspath(os.path.expanduser(dump_file))
36      # XXX Volatility not recognizes the input file when putting "{0}" ?
37      cmd += ' -f {0} {1} {2}'.format(dump_file, extra_params, plugin_name)
38      print(f'[*] Executing cmd: {cmd} ...', end='')
39      _completed_process = subprocess.run(cmd.split(' '), capture_output=
            True, close_fds=True)
40      print(' done!')
41      # check error code
42      if _completed_process.returncode != 0:
43          print('Execution of command "{0}" finished with return code {1}'.
                format(cmd, _completed_process.returncode))
44          print('[-] ERROR found: {0} '.format(_completed_process.stderr.
                decode("utf-8")))
45          return None, _completed_process.returncode
46
47      return _completed_process.stdout.decode("utf-8"), _completed_process.
            returncode
48
49  def print_data(json: dict, keys: list):
50      return
51
52  def JSONstr_to_DataFrame(output: str) -> pd.DataFrame:
53      _json = json.loads(output)
54      _list = [item for item in _json['rows']]
55      # set rows
56      df = pd.DataFrame.from_dict(_list)
57      # set column names
58      df.columns = _json['columns']
59      return df
60
61  if __name__ == "__main__":
62      load_cfg_file()
63      #print(create_tmpdir())
64      output, code = execute_vol_plugin("pslist", "~/Desktop/alina1G.elf",
            profile="Win7SP1x86", extra_params="--output=json")
65      df = JSONstr_to_DataFrame(output)
66      print(df[['Name', 'PID']])
67      print(df.loc[df['Name'].str.contains("ALINA", case=False)])
```

