Automatización de tareas de análisis con Volatility

En este laboratorio vas a introducirte en el desarrollo de procesos de automatización de tareas de análisis con Volatility. Para ello, usaremos el lenguaje Python3 para ejecutar un *workflow* de tareas de análisis. El objetivo de este laboratorio es describir las típicas nociones que necesitas conocer para lograr realizar la integración de tareas de análisis automática de una manera óptima.

Desde un punto de vista de ingeniería del software recuerda que el software ha de ser cuanto más desacoplado y modular mejor, puesto que permite una mejor reutilización del código y poder evaluar y mejorar componentes del sistema de manera independiente. A continuación encontrarás diversas trozos de código que se encargan de realizar tareas específicas y necesarias para disponer de un *workflow* de tareas de análisis.

1. Lectura de archivos de configuración

Para poder automatizar tareas de análisis con Volatility, existen dos elementos principales que se han de considerar: la ruta a la propia aplicación de Volatility y, en caso de que se vaya a trabajar con Python2, la ruta a la aplicación de Python2. La lectura de estas rutas se puede realizar mediante un fichero de configuración, usando el paquete de Python3 `configparse`:

```
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'])
```

El código anterior permite leer desde un fichero de configuración las rutas de ambas aplicaciones. Por defecto, el nombre del fichero de configuración se ha establecido a «`config.ini`». Este fichero ha de tener un formato similar al que se muestra en el Código 1, adaptado debidamente a las rutas de las aplicaciones en el sistema particular.

Código 1: Contenido del fichero «`config.ini`» para conocer las rutas de Volatility y Python2 en la máquina de análisis.

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

2. Creación de directorio de trabajo temporal

Normalmente, durante la ejecución de tareas de análisis se crean una serie de ficheros intermedios que no son de interés para el análisis. Lo mejor es disponer de un directorio temporal donde se creen estos ficheros, eliminándolo al acabar la ejecución del programa. Para ello, puede hacerse uso del paquete de Python3 `tempfile`:

```
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
```

Este trozo de código devolverá una ruta de un directorio creado con un nombre diferente en cada ejecución (el nombre depende del momento actual) y ubicado en el directorio temporal de trabajo.

3. Ejecución de Volatility

El siguiente trozo de código permite ejecutar un plugin de Volatility2 sobre un volcado dado por parámetro, permitiendo además especificar el perfil del volcado y los parámetros adicionales que sean necesarios. El primer y segundo parámetro son parámetros posicionales, siendo el nombre del plugin y el nombre del volcado respectivamente. El resto de parámetros son parámetros de palabra clave.

Este código ejecuta el proceso de Python2.7 junto con Volatility2, y devuelve la salida del comando y el código de ejecución devuelto en caso de una ejecución exitosa, o `None` y el código de ejecución en caso de error.


```
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
```

Por ejemplo, la siguiente línea de código permite ejecutar Volatility2 y el plugin `pslist` sobre el volcado de nombre «`alina1G.elf`»:

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

Si se quisiera obtener la salida en formato JSON (aprovechándose de la salida unificada de Volatility), es tan simple como añadir el parámetro `extra_params` a la invocación anterior:

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

Para finalizar, la salida de Volatility puede transformarse a una `DataFrame` de `pandas` con objeto de facilitar el procesado automático. Esto puede conseguirse de manera sencilla, incorporando al código la siguiente función que se encarga de procesar una cadena (que realmente será un diccionario) recibida por parámetro y transformarla en una unidad de `DataFrame`:


```
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
```

La ventaja de disponer de un `DataFrame` es que permite realizar operaciones típicas de extracción de columnas de interés muy fácilmente:

```
>>> 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
```


Del mismo modo, se puede también realizar filtrados específicos, según el valor de alguna columna en particular:

```
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. Resumen

En resumen, Python permite una automatización de tareas de análisis relativamente sencilla puesto que dispone de multitud de paquetes y librerías que facilitan el trabajo. Por ejemplo, puede invocarse el plugin `dlldump` para extraer las bibliotecas DLL de un determinado proceso (que cumpla unos determinados requisitos) y después analizarlas mediante el software `ClamAV`[1] o realizar un análisis estático más detallado mediante `pefile`, `Capstone`, o herramientas del estilo. En el Código 2 se encuentra todo el código utilizado como ejemplo a lo largo de este taller.

Código 2: Código de ejemplo de automatización de tareas de análisis (fichero «`automated.py`»).

```
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] Por ejemplo, puede usarse el paquete `pyClamd` (disponible en `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)])
```

