Desarrollo de herramientas de análisis propias

En este laboratorio vas a introducirte en el desarrollo de plugins con Volatility. Para ello, vas a hacer uso de la máquina virtual proporcionada por el profesor junto con uno de los volcados de memoria. Todo este material adicional lo tienes disponible en la página web de este taller formativo: `http://webdiis.unizar.es/~ricardo/sbc-2021/`. En el primer taller de este curso se ha explicado Volatility en mayor profundidad, presentando todos los plugins que vienen por defecto y pueden resultar de utilidad para la detección de software dañino en volcados de memoria. En este segundo taller se va a profundizar en el desarrollo de herramientas propias de análisis con Volatility. En concreto, se va a explicar el proceso de desarrollo de un plugin tanto en Volatility 2 como en Volatility 3.

1. Plugins de Volatility

El entorno de análisis de volcados de memoria Volatility funciona mediante *plugins*. Por defecto, Volatility viene con una serie de plugins oficiales (como `pslist`, `moddump`, o `dlllist`, por mencionar algunos). Además de los plugins oficiales, podemos desarrollar herramientas de análisis propias que funcionen como plugins de Volatility.

Volatility es un entorno desarrollado con Python. Dependiendo de la versión de Volatility, se trabaja con Python 2 o con Python 3. Por tanto, es necesario que tengas conocimientos de programación con Python para poder desarrollar tu propia herramienta de análisis. Se recomienda acudir al repositorio oficial de Python en caso de necesitar profundiza más en el lenguaje de Python (`https://docs.python.org/3/tutorial/`).

Recuerda que la ejecución de un plugin de Volatility 2 se realiza con el siguiente comando:

```
python vol.py -f MEMDUMP --profile=PROFILE PLUGINNAME
```

Con el parámetro `-f` se especifica el volcado de memoria a analizar y con `--profile` se especifica el perfil del volcado de memoria. El perfil tiene que coincidir con el sistema operativo del que proviene el volcado de memoria, dado que define dónde se encuentran las estructuras internas necesarias por Volatility para realizar el análisis forense de memoria. Por último, aparece el nombre del plugin que se quiere utilizar.

En el caso de Volatility 3, no es necesario especificar el perfil. Por contra, los nombres de los plugins,han cambiado ligeramente ya que ahora hay que especificar a qué sistema operativo nos estamos refiriendo. Por ejemplo, el plugin `pslist` puede ser `linux.pslist.PsList`, `mac.pslist.PsList` o `windows.pslist.PsList`.

Otro de los grandes cambios introducidos en Volatility 3 es que requiere disponer del fichero de símbolos del sistema operativo, en un formato especial JSON, para ser capaz de interpretar un volcado de memoria. Por defecto, Volatility 3 trata de localizar el fichero localmente. Si no lo encuentra, en el caso de Windows lo descarga del servidor de símbolos de Microsoft y lo

convierte al formato JSON apropiado. Estos archivos JSON residen, por defecto, en la carpeta volatility » symbols (dentro de un directorio para cada sistema operativo; windows, mac, o linux). Se recomienda la lectura de la documentación oficial de Volatility 3 en `https://volatility3.readthedocs.io/en/latest/symbol-tables.html` para saber más sobre este tema.

Instalación manual de símbolos de Windows en Volatility 3

Para saber cómo solucionar de manera manual posibles errores durante el análisis de volcados de memoria con Volatility 3, vamos a realizar la prueba con el volcado de ALINA disponible en `http://webdiis.unizar.es/~ricardo/sbc-2021/adicional/volcados/alina1G.elf.tar.gz`. Antes de continuar, será necesario instalar dos paquetes más en tu entorno de Python3:

```
pip3 install pdbparse
pip3 install jsonschema
```

Una vez instalados correctamente, podemos continuar. Como prueba de concepto vamos a ejecutar el plugin `windows.info`:

```
python3 vol.py -f ~/volcados/alina1G.elf windows.info
Volatility 3 Framework 1.0.1
WARNING  volatility3.framework.plugins: Automagic exception occurred:
    volatility3.framework.exceptions.InvalidAddressException: Offset
    outside of the buffer boundaries

Unsatisfied requirement plugins.Info.nt_symbols: Windows kernel symbols

A symbol table requirement was not fulfilled.  Please verify that:
    You have the correct symbol file for the requirement
    The symbol file is under the correct directory or zip file
    The symbol file is named appropriately or contains the correct banner

Unable to validate the plugin requirements: ['plugins.Info.nt_symbols']
```

Como puedes observar, ocurre un error durante la ejecución informando acerca de que ha ocurrido algún problema con la tabla de símbolos. Si ejecutamos el comando de nuevo con algo más de información de depuración (parámetro `-vvv`):

```
python3 vol.py -f ~/volcados/alina1G.elf -vvv windows.info
Volatility 3 Framework 1.0.1
INFO     volatility3.cli: Volatility plugins path: ['/Users/ricardo/
    volatility3/volatility3/plugins', '/Users/ricardo/volatility3/
    volatility3/framework/plugins']
[... omitido ...]
INFO     volatility3.framework.symbols.windows.pdbconv: Download PDB file
    ...
DEBUG    volatility3.framework.symbols.windows.pdbconv: Attempting to
    retrieve http://msdl.microsoft.com/download/symbols/ntkrnlmp.pdb/00625
    D7D36754CBEBA4533BA9A0F3FE22/ntkrnlmp.pdb
Level 9  volatility3.framework.configuration.requirements: Symbol table
    requirement not yet fulfilled: plugins.Info.nt_symbols0F3FE22/ntkrnlmp
    .pdb
WARNING  volatility3.framework.plugins: Automagic exception occurred:
    volatility3.framework.exceptions.InvalidAddressException: Offset
    outside of the buffer boundaries
```

Se observa que existe un error tras la descarga del PDB del servidor de Microsoft. Para realizar una instalación de los símbolos manual de Windows, primero hay que descargar el citado fichero PDB (por ejemplo, con `wget`):

```
wget http://msdl.microsoft.com/download/symbols/ntkrnlmp.pdb/00625
    D7D36754CBEBA4533BA9A0F3FE22/ntkrnlmp.pdb
--2021-07-14 11:43:23--  http://msdl.microsoft.com/download/symbols/
    ntkrnlmp.pdb/00625D7D36754CBEBA4533BA9A0F3FE22/ntkrnlmp.pdb
Resolving msdl.microsoft.com (msdl.microsoft.com)... 204.79.197.219
Connecting to msdl.microsoft.com (msdl.microsoft.com)
    |204.79.197.219|:80... connected.
[... omitido...]
HTTP request sent, awaiting response... 200 OK
Length: 6884352 (6.6M) [application/octet-stream]
Saving to: 'ntkrnlmp.pdb'

2021-07-14 11:43:38 (504 KB/s) - 'ntkrnlmp.pdb' saved [6884352/6884352]
```

Una vez descargado, podemos ejecutar el siguiente comando desde la carpeta principal de Volatility 3. El valor que acompaña al parámetro `-g` es el identificador que se ha usado en la dirección de descarga.

```
PYTHONPATH="." python3 volatility3/framework/symbols/windows/pdbconv.py -
    f ../ntkrnlmp.pdb -g 00625D7D36754CBEBA4533BA9A0F3FE22
```

La ejecución de este comando creará un fichero con nombre `00625D7D36754CBEBA4533BA9A0F3FE2-2.json.xz` en la carpeta donde se ha ejecutado. Ahora, este fichero comprimido hay que llevarlo a la carpeta `volatility3/symbols/windows/ntkrnlmp.pdb/`:

```
mv 00625D7D36754CBEBA4533BA9A0F3FE2-2.json.xz volatility3/symbols/windows
    /ntkrnlmp.pdb/.
```

Tras esto, la ejecución del plugin `windows.info` ahora será correcta:

```
[11:49:00] ricardo:volatility3 git:(develop*) $ python3 vol.py -f ~/volcados/alina1G.elf windows.info
Volatility 3 Framework 1.0.1
Progress:  100.00               PDB scanning finished
Variable        Value

Kernel Base     0x82805000
DTB     0x185000
Symbols file:///Users/ricardo/volatility3/volatility3/symbols/windows/ntkrnlmp.pdb/00625D7D36754CBEBA4533BA9A0F3FE2-2.json.xz
Is64Bit False
IsPAE   False
primary 0 WindowsIntel
memory_layer    1 Elf64Layer
base_layer      2 FileLayer
KdDebuggerDataBlock     0x82926c28
NTBuildLab      7601.17514.x86fre.win7sp1_rtm.10
CSDVersion      1
KdVersionBlock  0x82926c00
Major/Minor     15.7601
MachineType     332
KeNumberProcessors      1
SystemTime      2019-09-21 12:07:14
NtSystemRoot    C:\Windows
NtProductType   NtProductWinNt
NtMajorVersion  6
NtMinorVersion  1
PE MajorOperatingSystemVersion  6
PE MinorOperatingSystemVersion  1
PE Machine      332
PE TimeDateStamp        Sat Nov 20 08:42:46 2010
```

2. Desarrollo de plugins en Volatility 2

En esta sección se va a detallar cómo desarrollar un plugin para Volatility 2. Lo primero es crear una carpeta donde se desarrollará el plugin. Vamos a crear una carpeta con el nombre `myFirstPlugin` y dirigirnos a ella:

```
cd ~
mkdir myFirstPlugin
cd myFirstPlugin
```

Allí, vamos a crear un fichero llamado `myplugin.py` que será el fichero de código fuente del plugin que vamos a crear. Puedes usar el editor de texto de tu elección para ello, aunque se recomienda que sea algún tipo de entorno de programación para facilitarte la programación en Python (coloreado de sintaxis, tabulación correcta, etc.). En la máquina virtual proporcionada en este curso se ha instalado Visual Studio Code.

La estructura mínima de un plugin es la siguiente:

```
1 import volatility.plugins.common as common
2 class MyPlugin(common.AbstractWindowsCommand):
3     ''' My plugin '''
4
5     def render_text(self, outfd, data):
6         print "Hola, mundo!"
```

La primera línea de código está importando la librería `volatility.plugins.common`, necesaria en todos los plugins de Volatility 2. Después, en la línea 2 se está definiendo una clase que será la clase que instancia Volatility cuando se ejecuta el plugin. Esta clase tiene que heredar de una de las subclases de `common.Command`. En este caso, como vamos a desarrollar un plugin para Windows, se ha escogido la clase `common.AbstractWindowsCommand`. La cadena que aparece a continuación (línea 3) es la documentación que se escribirá por pantalla cuando se ejecute el plugin desde Volatility con la opción `-h` (o `--help`).

En la línea 5 se ha definido el método `render_text`. Este método es el que se encarga de presentar los resultados en formato de texto plano. Recibe dos parámetros:

- `outfd`, que es el descriptor de fichero en el que Volatility escribirá el texto. Por defecto, es la salida estándar, aunque se puede proporcionar un fichero como salida. El plugin puede presentar los resultados en diferentes formatos, como JSON o HTML. Entraremos en detalles más tarde.
- `data`, que contiene la información que haya recogido el plugin durante su análisis lista para ser mostrada.

Este plugin que acabamos de realizar simplemente escribirá "Hola, mundo!". Para ejecutarlo, simplemente desde la línea de comandos de Volatility hay que usar el parámetro `--plugins` con la ruta absoluta adonde se encuentra el plugin propio:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f ~/volcados/
    alina1G.elf myplugin
Volatility Foundation Volatility Framework 2.6.1
Hola, mundo!
```

Como puedes observar, se ha escrito la cadena de "Hola, mundo!" por pantalla. Vamos a añadir algo más de código al plugin para seguir mejorándolo. Considera ahora el siguiente código:


```
1  import volatility.plugins.common as common
2  import volatility.utils as utils
3  import volatility.win32 as win32
4
5  class MyPlugin(common.AbstractWindowsCommand):
6      ''' My plugin '''
7
8      def calculate(self):
9          addr_space = utils.load_as(self._config)
10         tasks = win32.tasks.pslist(addr_space)
11         return tasks
12
13     def render_text(self, outfd, data):
14         for task in data:
15             outfd.write("{!s}\n".format(str(task)))
```

Como puedes observar, se han añadido una serie de librerías necesarias para la correcta ejecución del plugin. Además, se ha añadido un nuevo método, `calculate`. Este método será llamado por Volatility tras la apertura y carga del volcado de memoria y se encargará de realiza el trabajo necesario de análisis. En concreto, lo que devuelve este método se usará para darle valor a ese parámetro `data` que recibe posteriormente el método `render_text`.

Este método `calculate` lo primero que hace es cargar el espacio de direcciones del volcado en una variable `addr_space`, a través de la llamada a `utils.load_as` con el objeto `self_config` pasado como parámetro. Este objeto de configuración contiene datos de la ejecución actual de Volatility, como por ejemplo los parámetros pasados por argumento. Esta variable `addr_space` se pasa como argumento a `win32.tasks.pslist` para obtener en la variable `tasks` la lista de todos los procesos que se estaban ejecutando en el volcado de memoria bajo análisis.

Esta variable `tasks` se devuelve y se acabará recibiendo por el método `render_text`, donde se itera sobre ella escribiéndola por pantalla.

Si probamos ahora este nuevo plugin, observaremos por pantalla algo similar a:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f ~/volcados/
    alina1G.elf --profile=Win7SP1x86 myplugin
Volatility Foundation Volatility Framework 2.6.1
2216900432
2231980416
2240262192
2240302400
[... omitido ...]
```

Cada elemento de la variable `tasks` es en realidad un objeto de `volatility.plugins.overlays.windows.windows._EPROCESS` (puedes consultar más sobre esta clase en la documentación oficial: `https://volatilityfoundation.github.io/volatility/d6/d38/classvolatility_1_1plugins_1_1overlays_1_1windows_1_1windows_1_1___e_p_r_o_c_e_s_s.html`). Podemos refinar ese método `render_text` para escribir algo más de información de interés de cada tarea. Por ejemplo:

```
13     def render_text(self, outfd, data):
14         for task in data:
15             outfd.write("{} {} {}\n".format(task.UniqueProcessId, task.ImageFileName, task
```


Si ejecutamos ahora el plugin de nuevo:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f ~/volcados/
    alina1G.elf --profile=Win7SP1x86 myplugin
Volatility Foundation Volatility Framework 2.6.1
4 System False
268 smss.exe False
348 csrss.exe False
384 wininit.exe False
[... omitido ...]
```

Como puedes observar, la información que se muestra ahora es más indicativa: hemos impreso el identificador del proceso, el nombre del proceso y un valor booleano que nos indica si se trata de un proceso WoW64 o no.

2.1. Salida unificada

Esto que acabamos de realizar es un plugin sencillo, y era la forma de trabajar con Volatility antes de la versión 2.5. A partir de esta versión, se introdujo la salida unificada (del inglés, *unified output*), que permite a un usuario usar un mismo plugin sin preocuparse del formato de salida: el usuario puede querer la salida en formato CSV, JSON, o incluso SQLite.

Para ello, necesitamos introducir un método `unified_output` en nuestro código. Este método tiene que devolver un objeto `TreeGrid`, que es un objeto cuyo constructor requiere dos parámetros: una lista de tuplas que definen un título y tipo de los datos que se van a devolver y un método que será el encargado de generar los datos.

Vamos a transformar el código del plugin que hemos realizado anteriormente para incorporar una salida unificada. Para ello, vamos a añadirle un método `unified_output` como sigue:

```
1 from volatility.renderers import TreeGrid
2
3     def unified_output(self, data):
4
5         return TreeGrid([
6             ("PID", int),
7             ("Image", str),
8             ("WoW64", int)],
9             self.generator(data))
```

El código `unified_output` está definiendo un `TreeGrid` donde la lista de tuplas está compuesta de tres columnas (`"PID"` de tipo entero, `Image` de tipo cadena y `WoW64` de tipo entero) y el método `generator`, que será un método que tiene que devolver un objeto conformado por un entero, una cadena y otro entero. Este método puede ser similar a:

```
1     def generator(self, data):
2         for task in data:
3             yield (0, [
4                 int(task.UniqueProcessId),
5                 str(task.ImageFileName),
6                 int(task.IsWow64)
7             ])
```

Si ahora ejecutamos la salida, veremos que es muy similar a la que teníamos antes aunque con un formato más cuidado:


```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f ~/volcados/
    alina1G.elf --profile=Win7SP1x86 myplugin
Volatility Foundation Volatility Framework 2.6.1
PID   Image           WoW64
   4 System              0
 268 smss.exe            0
 348 csrss.exe           0
 384 wininit.exe         0
[... omitido ...]
```

Por último, la ventaja de tener la salida unificada es que el desarrollador puede olvidarse de los formato de salida, delegando en el propio Volatility para ello. Por ejemplo, si al ejecutar Volatility añadimos el parámetro `--output=json` para indicarle a Volatility que nos interesa la salida en formato JSON, obtendremos una salida ya formateada en dicho formato:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f ~/volcados/
    alina1G.elf --profile=Win7SP1x86 myplugin --output=json
Volatility Foundation Volatility Framework 2.6.1
{"rows": [[4, "System", 0], [268, "smss.exe", 0], [348, "csrss.exe", 0],
    [384, "wininit.exe", 0], [392, "csrss.exe", 0], [432, "winlogon.exe",
    0], [476, "services.exe", 0], [484, "lsass.exe", 0], [492, "lsm.exe",
    0], [596, "svchost.exe", 0], [660, "VBoxService.ex", 0], [712, "
    svchost.exe", 0], [764, "svchost.exe", 0], [884, "svchost.exe", 0],
    [928, "svchost.exe", 0], [988, "audiodg.exe", 0], [1096, "svchost.exe
    ", 0], [1228, "svchost.exe", 0], [1308, "spoolsv.exe", 0], [1344, "
    svchost.exe", 0], [1448, "svchost.exe", 0], [1864, "taskhost.exe", 0],
     [1924, "dwm.exe", 0], [1940, "explorer.exe", 0], [316, "VBoxTray.exe
    ", 0], [1876, "SearchIndexer.", 0], [320, "SearchProtocol", 0], [1128,
     "SearchFilterHo", 0], [1828, "ALINA_CJLXYJ.e", 0]], "columns": ["PID
    ", "Image", "WoW64"]}
```

2.2. Parámetros

Los plugins pueden tener parámetros para ampliar su funcionalidad. La definición de parámetros se realiza en el método constructor `__init__()` del plugin. Se pueden definir los parámetros que deseemos, aunque estamos limitados en cuanto a la elección de opciones para los parámetros: no se pueden usar los mismos que por defecto ya usa Volatility (por ejemplo, `-p` o `-h`).

A modo de ejemplo, vamos a definir un nuevo parámetro para indicar que nos interesa exclusivamente aquellos procesos que sean WoW64:

```
1     def __init__(self, config, *args, **kwargs):
2         common.AbstractWindowsCommand.__init__(self, config,
3                                                *args, **kwargs)
4         self._config.add_option('ONLYWOW64', short_option = 'W',
5             default = False, help = 'Only show WoW64 processes',
6             action = 'store_true')
```

La primera línea del constructor está invocando al constructor de la clase madre. Después, estamos añadiendo una nueva opción (parámetro) mediante el método `add_option` de `self._config`. La sintaxis es muy similar al paquete `argparse` de Python. Los parámetros que recibe este método son los siguientes:

- El primer parámetro, `'ONLYWOW64'`, es el nombre largo que queremos darle al parámetro.
- Opcionalmente, podemos añadirle un nombre corto mediante el parámetro `short_option`. En este caso, se ha seleccionado `'W'`. Recuerda que ni el nombre largo ni el nombre corto pueden coincidir con los parámetros definidos por defecto por Volatility.
- Con `default=None` especificamos el valor por defecto que tendrá este parámetro si no viene especificado por el usuario. En este caso, valor `None`.
- Con el parámetro `help` se especifica el texto de ayuda del parámetro que aparecerá en la terminal cuando se invoque al plugin con el parámetro de ayuda (`-h` o `--help`).
- El parámetro `action = 'store_true'` indica que si el parámetro viene indicado, almacenará el valor `True`. El valor del parámetro `action` puede ser `'store'` cuando se quiere guardar el valor que se proporcione, `'store_false'` si se desea almacenar el valor `False` o `'append'` si se permiten múltiples parámetros (en este caso, se construirá una lista con todos los parámetros).

Vamos a modificar ahora el código del plugin para que, después de conseguir los procesos en ejecución, si está el parámetro `-W` o `--onlywow64` se quede con el subconjunto de tareas que son procesos WoW64. Para ello, vamos a hacer uso de los generadores de Python y definir un nuevo método, `onlyWow64`, que se encargue de recorrer el generador original y se quede con aquellas tareas de interés:

```
1     def onlyWow64(self, tasks):
2         for task in tasks:
3             if task.IsWow64:
4                 yield task
5
6     def calculate(self):
7         addr_space = utils.load_as(self._config)
8         tasks = win32.tasks.pslist(addr_space)
9
10        wow64 = self._config.ONLYWOW64
11        if wow64:
12            tasks = self.onlyWow64(tasks)
13
14        return tasks
```

Con todo esto, ya podemos ejecutar el plugin de nuevo, y esta vez con el nuevo parámetro para observar su comportamiento:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f ~/volcados/
    alina1G.elf --profile=Win7SP1x86 myplugin --output=json --onlywow64
Volatility Foundation Volatility Framework 2.6.1
{"rows": [], "columns": ["PID", "Image", "WoW64"]}%
```

Como se muestra, en este caso la salida no nos muestra ningún resultado puesto que este volcado no contenía ningún proceso que fuera WoW64.

2.3. Uso de otros plugins

En el repositorio oficial de Volatility puedes encontrar todos los módulos que se han desarrollado por parte de la comunidad: `https://github.com/volatilityfoundation/community`.

Cuando necesites ampliar la funcionalidad de análisis de Volatility, es altamente recomendable que primero intentes localizar si ya existe algún plugin desarrollado que haga totalmente (o parcialmente) lo que necesitas.

En caso de que necesites alguna funcionalidad de algún plugin ya existe, puedes aprovecharla. Lo más sencillo en estos casos es que estudies el código fuente del plugin que te interesa, entiendas cómo instanciarlo (qué parámetros necesitas para construir un objeto), lo configures y lo ejecutes.

Normalmente requerirás crear un objeto de tipo `ConfObject`, rellenarlo de manera adecuada con los parámetros del plugin y luego usarlo en el constructor del plugin. Después, para ejecutarlo tendrás que usar el método `execute` del objeto. Finalmente, recuerda liberar la memoria una vez finalices con el uso del objeto (instrucción `del` de Python).

3. Desarrollo de plugins en Volatility 3

Ahora, vamos a hacer un proceso similar pero en este caso con la versión 3 de Volatility. En Volatility 3, la clase de nuestro plugin tiene que heredar de `PluginInterface`. Volatility encuentra todos los plugins que estén en los directorios de plugin y los importa, usando todos aquellos que hereden de esta clase `PluginInterface`.

Dirígete a la carpeta volatility3 » plugins » windows dentro del directorio raíz de Volatility 3 y crea un fichero con nombre `myplugin.py`.

```
1  from typing import List
2
3  from volatility3.framework import renderers, interfaces
4  from volatility3.framework.configuration import requirements
5  from volatility3.framework.interfaces import plugins
6  from volatility3.plugins.windows import pslist
7
8  class MyFirstPlugin(plugins.PluginInterface):
9      _required_framework_version = (1, 0, 0)
```

Además de las bibliotecas importadas, se ha definido que se requiere la versión mínima de 1.0.0 de Volatility 3. A continuación hay que definir los requisitos, que definen aquellas variables que son necesarias para el plugin pueda funcionar correctamente. Si un requisito se define como opcional, no es necesario entonces darle valor. Esta definición se realiza en el método `get_requirements`.

```
1  @classmethod
2      def get_requirements(cls) ->
3                      List[interfaces.configuration.RequirementInterface]:
4          return [
5              requirements.TranslationLayerRequirement(name = 'primary',
6                              description = 'Memory layer for the kernel',
7                              architectures = ["Intel32", "Intel64"]),
8              requirements.SymbolTableRequirement(name = "nt_symbols",
9                              description = "Windows kernel symbols"),
10             requirements.BooleanRequirement(name = 'onlywow64',
11                             description = "Only show WoW64 processes",
12                             default = False,
13                             optional = True)
14         ]
```

Observa que se trata de un método de clase, puesto que se invoca antes de instanciar el objeto concreto del plugin (la información que proporciona es necesaria para instanciar el plugin). En

este caso se está definiendo que el plugin trabajará sólo con volcados de arquitecturas de Intel de 32 o 64 bits, que necesita los símbolos del núcleo de Windows y como parámetro opcional se ha definido `onlywow64`, con valor por defecto a `False`.

En Volatility 3 hay que definir un método `run`, que será el que llame Volaility tras realizar la carga del volcado de memoria. Este método devuelve un objeto de tipo `TreeGrid`, que al igual que en Volatility 2 sirve para facilitar la obtención de un formato de salida determinado por el usuario por parámetro. Vamos a añadir el siguiente código:

```
1     def run(self):
2         tasks = pslist.PsList.list_processes(self.context,
3                                              self.config['primary'],
4                                              self.config['nt_symbols'])
5         wow64 = self.config['onlywow64']
6         if wow64:
7             tasks = self.onlyWow64(tasks)
8
9         return renderers.TreeGrid([("PID", int), ("Image", str),
10                                   ("WoW64", int)], self._generator(tasks))
```

Observa que el método `run` realiza lo mismo que en el plugin desarrollado para Volatility 2, pero adaptándolo a Volatility 3. Básicamente, cambia la forma de recuperar la lista de procesos (se está recuperando a través del propio plugin `pslist`, línea 2). Después, se comprueba que esté definido el parámetro `--onlywow64` para, en tal caso, filtrar los procesos para quedarse con aquellos que sean procesos WoW64 (líneas 5 a 7). Por último, en la línea 9 se está creando el objeto `TreeGrid`, que sigue la misma definición que en Volatility2: necesita la tupla de valores (cadenas y tipo) y como segundo parámetro, el generador.

Los dos métodos que faltan por definir, `onlyWow64` y `generator`, serían como se muestra a continuación. Observa que cambia la forma de imprimir el campo `ImageFileName` de la tarea y la forma de acceder al booleano que define si es o no un proceso WoW64.

```
1     def _generator(self, data):
2         for task in data:
3             yield (0, [
4                     int(task.UniqueProcessId),
5                     task.ImageFileName.cast("string",
6                             max_length = task.ImageFileName.vol.count,
7                             errors = 'replace'),
8                     int(task.get_is_wow64())
9                 ])
10
11    def onlyWow64(self, tasks):
12        for task in tasks:
13            if task.get_is_wow64():
14                yield task
```


Con todo esto, ya podemos proceder a la ejecución:

```
python3 vol.py -f ~/volcados/alina1G.elf windows.myfirstplugin.
    MyFirstPlugin
Volatility 3 Framework 1.0.1
Progress:  100.00       PDB scanning finished
PID Image   WoW64

4   System  0
268 smss.exe    0
348 csrss.exe   0
384 wininit.exe 0
[... omitido ...]
```

Observa que el nombre del plugin es el nombre de la clase (`MyFirstPlugin`), antecedido por el directorio `windows` y el nombre del fichero (`myfirstplugin`) donde se encuentra definida. Como antes, si ejecutamos el plugin con el parámetro `--onlywow64` vemos que de nuevo no aparece ninguno:

```
python3 vol.py -f ~/volcados/alina1G.elf windows.myfirstplugin.
    MyFirstPlugin --onlywow64
Volatility 3 Framework 1.0.1
Progress:  100.00       PDB scanning finished
PID Image   WoW64
```

En cuanto a la salida unificada, en el caso de Volatility 3 ha cambiado el parámetro a `-r`. Por ejemplo, si se desea la salida formateada en formato JSON:

```
python3 vol.py -f ~/volcados/alina1G.elf -r json windows.myfirstplugin.
    MyFirstPlugin
Volatility 3 Framework 1.0.1
Progress:  100.00       PDB scanning finished
[
  {
    "Image": "System",
    "PID": 4,
    "WoW64": 0,
    "__children": []
  },
  {
    "Image": "smss.exe",
    "PID": 268,
    "WoW64": 0,
    "__children": []
  },
  {
    "Image": "csrss.exe",
    "PID": 348,
    "WoW64": 0,
    "__children": []
  },
[... omitido ...]
}
```

