34th Jyväskylä Summer School
Memory Forensics for Incident Response
University of Jyväskylä (Finland)

Hands-On Guide to Volatility Plugin Development for Incident Response

In this lab you will practice with plugin development with Volatility. To do this, you are going to make use of the virtual machine provided by the instructor along with one of the memory dumps. All this additional material is available on the web page of this training workshop: `https://webdiis.unizar.es/~ricardo/cyb1-jss-25`. In the first workshop of this course, Volatility has been explained in greater depth, presenting all the plugins that come by default and can be useful for detecting harmful software in memory dumps. In this third workshop, the development of our own analysis tools with Volatility will be delved into. Specifically, the process of developing a plugin in both Volatility 2 and Volatility 3 will be explained.

By the end of this lab, you will be able to develop custom Volatility plugins for Volatility 2 and Volatility 3, handle symbol files manually (when Volatility 3 is unable to locate them within the memory dump under analysis), and incorporate unified parameters and outputs.

Why Develop Your Own Plugins?

In real-world incident response and digital forensics investigations, analysts often face scenarios where Volatility's default set of plugins doesn't fully meet their investigative needs. While official and community plugins cover a wide range of forensic tasks, certain cases require custom analysis tools to address unique challenges. Some examples include:

- **Investigating new or uncommon malware families**: Newly discovered threats may use techniques not yet supported by existing plugins, requiring custom logic to extract relevant artifacts.
- **Focusing on specific forensic objectives**: Some investigations require specialized filtering, correlation of multiple data sources, or reporting in a custom format not available in standard tools.
- **Support for proprietary or non-standard environments**: Memory dumps from embedded devices, custom kernels, or modified operating systems may require unique analysis logic.
- **Automation of repetitive or complex analysis workflows**: A custom plugin can combine the functionality of several pre-built plugins, reducing the time and effort required for routine tasks.

- **Integration with internal analysis processes**: The results of a custom plugin can be formatted to feed directly into your organization's threat intelligence tools, reporting systems, or platforms.

By developing your own plugins, you gain the flexibility to tailor Volatility to emerging threats, unusual data structures, or specific operational requirements, making it a more powerful and targeted tool for your incident response team.

1 Volatility Plugins

The Volatility forensic analysis environment is powered by *plugins*. By default, Volatility comes with a number of official plugins (such as `pslist`, `moddump`, or `dlllist`, to name a few). In addition to the official plugins, we can develop our own analysis tools that work as Volatility plugins.

Volatility is an environment developed with Python. Depending on the version of Volatility, you work with Python 2 or Python 3. Therefore, you need to have programming knowledge with Python to be able to develop your own analysis tool. It is recommended to go to the official Python repository if you need to go deeper into the Python language (`https://docs.python.org/3/tutorial/`).

Remember that the execution of a Volatility 2 plugin is done with the following command:

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

The `-f` parameter specifies the memory dump to be analyzed and the `--profile` parameter specifies the memory dump profile. The profile has to match the operating system that the memory dump comes from, since it defines where the internal structures needed by Volatility to perform memory forensics are located. Finally, the name of the plugin that you want to use appears.

In the case of Volatility 3, it is not necessary to specify the profile. On the other hand, the names of the plugins have changed slightly since now we have to specify which operating system we are referring to. For example, the plugin `pslist` can be `linux.pslist.PsList`, `mac.pslist.PsList`, or `windows.pslist.PsList`.

Another of the big changes introduced in Volatility 3 is that it requires having the symbol file of the operating system, in a special JSON format, to be able to interpret a memory dump. By default, Volatility 3 tries to locate the file locally. If it doesn't find it, in the case of Windows it downloads it from the Microsoft Symbol Server and converts it to the appropriate JSON format. These JSON files reside, by default, in the volatility » symbols folder (within a directory for each operating system; windows, mac, or linux). Reading the official Volatility 3 documentation at `https://volatility3.readthedocs.io/en/latest/symbol-tables.html` is recommended to learn more about this topic.

Manual Installation of Windows Symbols in Volatility 3

In order to know how to manually solve possible errors during the analysis of memory dumps with Volatility 3, we are going to carry out the test with the ALINA dump available at `https://webdiis.unizar.es/~ricardo/cyb1-jss-25`. In case that the memory dump is not correctly identified by Volatility 3, you will need to install the kernel symbols by yourself.

If you are unable to know the version of the Windows machine in the ALINA dump (i.e., `windows.info` plugin fails to execute), you need to manually install the Windows kernel symbols corresponding to that memory dump. Before continuing, it will be necessary to install two more

packages in your Python3 environment:

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

Once installed correctly, we can continue. As a proof of concept let's run the `windows.info` plugin:

```
python3 vol.py -f /shared/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']
```

As you can see, an error occurs during execution informing about a problem with the symbol table. If we run the command again with some more debugging information (`-vvv` parameter):

```
python3 vol.py -f /shared/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']
[... snip ...]
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/00625D7D36754CBEBA4533BA9A0F3FE22/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
```

It is observed that there is an error after downloading the PDB from the Microsoft server. To install the Windows manual symbols, you must first download the aforementioned PDB file (for example, with `wget`):


```
wget http://msdl.microsoft.com/download/symbols/ntkrnlmp.pdb/00625D7D36754CBEBA4533BA9A0F3FE22/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.
[... snip ...]
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]
```

Once downloaded, we can run the following command from the main Volatility 3 folder. The value that accompanies the `-g` parameter is the identifier that has been used in the download address.

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

The execution of this command will create a file named «`00625D7D36754CBEBA4533BA9A0F3FE2-2.json.xz`» in the folder where it was executed. Now, this compressed file must be taken to the folder «`volatility3/symbols/windows/ntkrnlmp.pdb/`»:

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

After this, the execution of the `windows.info` plugin will now be successful:


```
[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 Plugin Development in Volatility 2

This section will detail how to develop a plugin for Volatility 2. The first thing is to create a folder where the plugin will be developed. Let's create a folder named «`myFirstPlugin`» and go to it:

```
1 cd ~
2 mkdir myFirstPlugin
3 cd myFirstPlugin
```

There, we are going to create a file called «`myplugin.py`» that will be the source code file of the plugin that we are going to create. You can use the text editor of your choice for this, although it is recommended that it be some kind of programming environment to make programming in Python easier for you (syntax coloring, correct tab, etc.). Visual Studio Code has been installed on the virtual machine provided in this course.

The minimum structure of a plugin is as follows:

```
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!"
```

The first line of code is importing the `volatility.plugins.common` library, which is necessary in all Volatility 2 plugins. Then, on line 2, a class is being defined that will be the class that Volatility instantiates when the plugin is executed. This class has to inherit from one of the `common.Command` subclasses. In this case, since we are going to develop a plugin for Windows, the class `common.AbstractWindowsCommand` has been chosen. The string below (line 3) is the

documentation that will be written to the screen when the plugin is run from Volatility with the `-h` option (or `--help`).

On line 5 the method `render_text` has been defined. This method is responsible for presenting the results in plain text format. It receives two parameters:

- `outfd`, which is the file descriptor to which Volatility will write the text. By default, this is standard output, although a file can be provided as output. The plugin can present the results in different formats, such as JSON or HTML. We will go into details later.
- `data`, which contains the information that the plugin has collected during its analysis ready to be displayed.

The plugin we just made will simply write "Hello, world!". To run it, simply from the Volatility command line you have to use the parameter `--plugins` with the absolute path where your own plugin is located:

```
1 python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f /shared/
↪    alina1G.elf myplugin
2 Volatility Foundation Volatility Framework 2.6.1
3 Hola, mundo!
```

As you can see, the string "Hello, world!" has been written to the screen. We are going to add some more code to the plugin to continue improving it. Now consider the following code:

```
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)))
```

As you can see, a series of necessary libraries have been added for the correct execution of the plugin. Also, a new method, `calculate`, has been added. This method will be called by Volatility after opening and loading the memory dump and will take care of doing the necessary analysis work. Specifically, what this method returns will be used to give value to that parameter `data` that the `render_text` method later receives.

The first thing this `calculate` method does is load the address space of the dump into a variable `addr_space`, by calling `utils.load_as` with the `self_config` object passed as parameter. This configuration object contains data from the current execution of Volatility, such as the parameters passed by argument. The variable `addr_space` is passed as an argument to `win32.tasks.pslist` to get in the variable `tasks` the list of all the processes that were running in the memory dump under analysis.

This `tasks` variable is returned and will end up being received by the `render_text` method, where it iterates over it, writing it to the screen.

If we try this new plugin now, we will see on the output something similar to:

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

Each element of the `tasks` variable is actually an object of `volatility.plugins.overlays.windows.windows._EPROCESS` (you can read more about this class in the official documentation: `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`). We can refine that `render_text` method to write some more information of interest to each task. For example:

```
    def render_text(self, outfd, data):
        for task in data:
            outfd.write("{} {} {}\n".format(task.UniqueProcessId, task.ImageFileName, task.IsWow64))
```

If we now run the plugin again:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f /shared/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
[... snip ...]
```

As you can see, the information shown now is more indicative: we have printed the process identifier, the process name and a boolean value that tells us if it is a WoW64 process or not.

2.1 Unified Output

What we just did is a simple plugin, and it was the way to work with Volatility before version 2.5. As of this version, *unified output* was introduced, which allows a user to use the same plugin without worrying about the output format: the user may want the output in CSV, JSON, or even SQLite.

To do this, we need to introduce a `unified_output` method in our code. This method has to return a `TreeGrid` object, which is an object whose constructor requires two parameters: a list of tuples that define a title and type of the data to be returned, and a method that will be responsible for generating the data.

We are going to transform the plugin code we made earlier to incorporate a unified output. To do this, we are going to add a `unified_output` method to it as follows:

```
from volatility.renderers import TreeGrid

    def unified_output(self, data):

        return TreeGrid([
            ("PID", int),
            ("Image", str),
            ("WoW64", int)],
            self.generator(data))
```

The `unified_output` code is defining a `TreeGrid` where the list of tuples is made up of three columns (`PID` of type integer, `Image` of type string, and `WoW64` of type integer) and the method `generator`, which will be a method that has to return an object made up of an integer, a string and another integer. This method can be similar to:

```
    def generator(self, data):
        for task in data:
            yield (0, [
                int(task.UniqueProcessId),
                str(task.ImageFileName),
                int(task.IsWow64)
            ])
```

If we now execute the output, we will see that it is very similar to the one we had before, although with a cleaner format:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f /shared/
↪   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
[... snip ...]
```

Finally, the advantage of having the unified output is that the developer can forget about the output formats, delegating to Volatility itself for it. For example, if when executing Volatility we add the parameter `--output=json` to tell Volatility that we are interested in the output in JSON format, we will obtain an output already formatted in that format:

```
python2 vol.py --plugins=/Users/ricardo/myFirstPlugin -f /shared/
↪   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 Parameters

Plugins can have parameters to extend their functionality. Parameter definition is done in the `__init__()` constructor method of the plugin. We can define the parameters we want, although we are limited in terms of the choice of options for the parameters: we cannot use the same ones that Volatility already uses by default (for example, `-p` or `-h`).

As an example, we are going to define a new parameter to indicate that we are only interested in those processes that are 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')
```

The first line of the constructor is calling the constructor of the parent class. Next, we are adding a new option (parameter) via the `add_option` method of `self._config`. The syntax is very similar to Python's `argparse` package. The parameters that this method receives are the following:

- The first parameter, `'ONLYWOW64'`, is the long name we want to give the parameter.
- Optionally, we can add a short name to it using the `short_option` parameter. In this case, `'W'` has been selected. Remember that neither the long name nor the short name can match the parameters defined by default by Volatility.
- With `default=None` we specify the default value that this parameter will have if it is not specified by the user. In this case, value `None`.
- With the parameter `help` you specify the help text of the parameter that will appear in the terminal when the plugin is invoked with the help parameter (`-h` or `--help`).
- The parameter `action = 'store_true'` indicates that if the parameter is specified, it will store the value `True`. The value of the `action` parameter can be `'store'` when you want to store the supplied value, `'store_false'` if you want to store the value `False` or `'append'` if multiple parameters are allowed (in this case, a list with all parameters will be constructed).

We are now going to modify the plugin code so that, after getting the running processes, if the `-W` or `--onlywow64` parameter is present, it will keep the subset of tasks that are WoW64 processes. To do this, we are going to make use of the Python generators and define a new method, `onlyWow64`, which is in charge of going through the original generator and keeping those tasks of interest:


```
    def onlyWow64(self, tasks):
        for task in tasks:
            if task.IsWow64:
                yield task

    def calculate(self):
        addr_space = utils.load_as(self._config)
        tasks = win32.tasks.pslist(addr_space)

        wow64 = self._config.ONLYWOW64
        if wow64:
            tasks = self.onlyWow64(tasks)

        return tasks
```

With all this, we can now run the plugin again, and this time with the new parameter to observe its behavior:

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

As shown, in this case the output does not show us any results since this dump did not contain any processes that were WoW64.

2.3 Use of Other Plugins

In the official Volatility repository you can find all the modules that have been developed by the community: `https://github.com/volatilityfoundation/community`. When you need to extend Volatility's analysis functionality, it is highly recommended that you first try to locate if there is already a developed plugin that does fully (or partially) what you need.

In case you need some functionality of a plugin that already exists, you can take advantage of it. The simplest thing in these cases is that you study the source code of the plugin you are interested in, understand how to instantiate it (what parameters you need to build an object), configure it, and run it.

Typically you will need to create an object of type `ConfObject`, populate it appropriately with the plugin parameters, and then use it in the plugin constructor. Then, to execute it you will have to use the object's `execute` method. Finally, remember to free the memory once you're done using the object (Python's `del` command).

3 Plugin Development in Volatility 3

Now, we are going to do a similar process but in this case with version 3 of Volatility. In Volatility 3, our plugin class has to inherit from `PluginInterface`. Volatility finds all the plugins that are in the plugin directories and imports them, using all those that inherit from this `PluginInterface` class.

Go to the volatility3 » plugins » windows folder inside the Volatility 3 root directory and create a file named «`myplugin.py`».

```
from typing import List

from volatility3.framework import renderers, interfaces
from volatility3.framework.configuration import requirements
from volatility3.framework.interfaces import plugins
from volatility3.plugins.windows import pslist

class MyFirstPlugin(plugins.PluginInterface):
    _required_framework_version = (1, 0, 0)
```

In addition to the imported libraries, it has been defined that the minimum version of 1.0.0 of Volatility 3 is required. Next, the requirements must be defined, which define those variables that are necessary for the plugin to work correctly. If a requirement is defined as optional, then it is not necessary to give it a value. This definition is done in the `get_requirements` method.

```
@classmethod
    def get_requirements(cls) ->
                    List[interfaces.configuration.RequirementInterface]:
        return [
            requirements.TranslationLayerRequirement(name = 'primary',
                            description = 'Memory layer for the kernel',
                            architectures = ["Intel32", "Intel64"]),
            requirements.SymbolTableRequirement(name = "nt_symbols",
                             description = "Windows kernel symbols"),
            requirements.BooleanRequirement(name = 'onlywow64',
                             description = "Only show WoW64 processes",
                             default = False,
                             optional = True)
        ]
```

Note that this is a class method, since it is called before instantiating the plugin's concrete object (the information it provides is needed to instantiate the plugin). In this case, it is being defined that the plugin will only work with 32 or 64-bit Intel architecture dumps, which needs the Windows kernel symbols and `onlywow64` has been defined as an optional parameter, with a default value of `False`.

In Volatility 3 you have to define a `run` method, which will be the one called by Volatility after loading the memory dump. This method returns an object of type `TreeGrid`, which, as in Volatility 2, is used to facilitate obtaining an output format determined by the user by parameter. Let's add the following code:


```
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))
```

Note that the `run` method does the same as in the plugin developed for Volatility 2, but adapting it to Volatility 3. Basically, it changes the way of retrieving the process list (it is being retrieved through the `plugin itself). pslist`, line 2). Then, it is checked that the parameter `--onlywow64` is defined to, in this case, filter the processes to keep those that are WoW64 processes (lines 5 to 7). Finally, on line 9 the `TreeGrid` object is being created, which follows the same definition as in Volatility2: it needs the tuple of values (strings and type) and the generator as the second parameter.

The two methods that remain to be defined, `onlyWow64` and `generator`, would be as shown below. Notice that it changes the way of printing the `ImageFileName` field of the task and the way of accessing the boolean that defines whether or not it is a WoW64 process.

```
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
```


With all this, we can proceed to the execution:

```
python3 vol.py -f /shared/alina1G.elf windows.myplugin.
↪   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
[... snip ...]
```

Notice that the plugin name is the name of the class (`MyFirstPlugin`), preceded by the directory `windows` and the name of the file (`myfirstplugin`) where it is defined. As before, if we run the plugin with the parameter `--onlywow64` we see that none appear again:

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

As for the unified output, in the case of Volatility 3 the parameter has been changed to `-r`. For example, if you want the output formatted in JSON format:


```
python3 vol.py -f /shared/alina1G.elf -r json windows.myplugin.
↪   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": []
  },
[... snip ...]
}
```

