Lab 4.2: Build a Volatility Plugin

Objectives

- Understand how volatility 3 plugins work
- Add a usefull new plugin to your arsenal that has not yet been ported to volatility 3

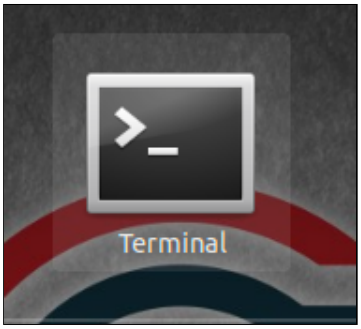
Volatility 2 had a nice plugin called psxview. It would extract process structure from various sources to identify processes that were hidden by the attacker. In this exercise we will write a mini version of that plugin for volatility 3. We will compare the output of pslist with psscan. That can be used to identify unlinked processes as discussed in the last sections. Volatility 3 is written in python 3. If you never wrote code in python before, don't worry, all blocks of code will be described and it's never too late to start coding. Matter of factly to make full use of volatility, python is mandatory.

I split the scaffold code into 3 sections and marked them with comments. That makes it easier to reference certain sections in the upcomming questions. Comment lines in python start with `#`

Preparation (Linux)

This part is completed in the **532 SIFT VM**

1. Begin by launching the **532 SIFT VM** and log in.
    - LOGIN = **sansforensics**
    - PASSWORD = **forensics**
2. Open a Terminal window



3. Navigate to the cases directory

```
cd /cases/c2
```

4. Open the plugin scaffold I prepared for you in a text editor

```
gedit /home/sansforensics/volatility3/volatility3/plugins/windows/minipsxview.py &
```

5. Test what the plugin does currently by pointing it at the `c2.vmem` image.

```
vol3 -f c2.vmem windows.minipsxview
```

Block 1 - Configuration

In this block almost everything is set up for you. As you can see, at first we import a number of functions from volatility. We won't reinvent the wheel. As we want to compare the data generated by the pslist command with the data generated by the psscan command, we might just use these plugins instead of recoding the logic into our plugin.

If you are lost, you can always also open the final stage of the plugin form the precooked folder.

```
gedit /cases/vol_plugin/precooked/minipsxview.py &
```

1. There seems to be one import missing. Can you spot it?

> **Hints**
>
> Can we already use psscan?
>
> > **Hints**
> >
> > Check line 7

> **Answer**
>
> While pslist is imported already, psscan is missing. Remember, we want to compare the output of these two modules. Just fix line 7 to look like this
>
> ```
> from volatility3.plugins.windows import pslist, psscan
> ```
>
> > **Discussion**
> >
> > Now we have everything imported that we need. I suggest to run the plugin in the terminal window after every change to fix errors as they occur rather than searching for them later.
> >
> > ```
> > vol3 -f c2.vmem windows.minipsxview
> > ```

Note on requirements

Volatility 3 plugins need to specify their requirements. This is done using a class method. This method is run before our plugin object is instantiated. Optional requirements can be used for parameters as well (like pslist accepts the --pid parameter to limit output to one pid only).

We have three requirements for our plugin, we need access to the primary layer (we dicussed layers of volatility a few minutes ago) and we'll only run that plugin on 32 bit and 64 bit Intel architectures.

We also require the **nt_symbols** tables to be loaded which effectively limits the plugin to windows systems.

© 2023 Mathias Fuchs

That pretty much concludes Block 1

Block 2 - Acuire the actual data

For our plugin to work, we need data from two plugins, psscan and pslist. Like discussed in class, psscan will identify processes by scanning for pool headers while pslist will walk the doubly linked list of **_EPROCESS** structures. This means, that psscan will find equally many or more processes as pslist. For example psscan might also show exited processes.

Block 2 consists of a run method. All we have to do for this section is getting the actual information from the two plugins into the variables in lines 36 and 38.

1. Can you fill in lines 36 and 38 so they get lists of **_EPROCESSES** (actually you'll get generators but that doesn't make a difference here)?

> **Hints**
>
> Check out how the two required plugins work an which methods they expose:
>
> - https://github.com/volatilityfoundation/volatility3/blob/develop/volatility3/framework/plugins/windows/pslist.py
> - https://github.com/volatilityfoundation/volatility3/blob/develop/volatility3/framework/plugins/windows/psscan.py

> **Answer**
>
> Below are the new lines 35 to 38
>
> ```
> # extract the process list leveraging the pslist plugin
> pslist_out =
> pslist.PsList.list_processes(self.context,self.config['primary'],self.config['nt_symbols'])
> # extract the process list leveraging the psscan plugin
> psscan_out =
> psscan.PsScan.scan_processes(self.context,self.config['primary'],self.config['nt_symbols'])
> ```
>
> > **Discussion**
> >
> > As volatility plugins should be self sufficient and don't rely on a central configuration store, when you call pslist or psscan, you need to pass on the context. In our case that means the required layer and the required symbol table. As you can see, the reason why our plugin requires these things is only to pass them on to the underlying plugin.
> >
> > For those of you who know python and want to play around, you can comment out lines 31 and 32 with a `#` character and for example build a loop to print out the structures in the variables `pslist_out` and `psscan_out`

Note on renderers

In line 43 we return a Tree grid renderer and propagate it with data. The renderer receives what fields the output data will have and the actual rendered data we will prepare in our renderer in Block 3. So volatility will call our plugin and expect rendered output like discussed in class.

Block 3 - Prepare the data for presentation

This is the block where most of the logic in our plugin happens. At this point we get the `pslist` and `psscan` data passed on. Now we need to process it. I prepared an empty array for the final data. It's called processes and needs to be filled. As psscan data will at least hold the same data as `pslist` but could have more data we will start propagating our array with that data first. The idea is to get the following fields per process:

- PID (int)
- Executable Name (str)
- is in psslist (bool)
- is in psscan (bool)

As you can see, that is the structure I already prepared the Treegrid renderer to support (line 43). We will first iterate through the `psscan` data and for every process found in that list set the `pslist` field to **False** and the `psscan` field to **True**. In a second round we iterate through the processes and check for every scanned process if it is also in `pslist`. If it is, we set the `pslist` field to true for this process.

1. Try to propagate the `processes` array with the psscan data.

   Hints

   You can iterate through lists and generators using `for` statements.

   Answer

   This simple `for` loop propagates the `process` array with the psscan data.

   ```
   for process in psscan_out:
       processes.append({"PID":process.UniqueProcessId,"name":process.ImageFileName.cast("string",
   max_length = process.ImageFileName.vol.count, errors = 'replace'),"psscan":True, "pslist":False})
   ```

   Discussion

   There are less expensive approaches to compare data than the one we use here, but as we have a very finite number of processes, we don't need to care too much about CPU consumption. We sacrifice a bit of performance for easier to read code.

2. Now you need to see whcih of the processes that now exist in the `process` array are also in the pslist data and set the flag for pslist to true if they are.

   ??? tip "Hints"git Try it with a nested `for` loop

© 2023 Mathias Fuchs

Answer

We just iterate through the pslist output as it is most likely the shorter list and then for every entry compare it to the already stored psscan results.

```
for listtask in pslist_out:
    for scantask in processes:
        if listtask.UniqueProcessId==scantask['PID']:
            scantask['pslist']=True
```

Finalizing Block 3

In the end we need to return a generator object for volatility to work. Technically we could also just use `print` to output our data but that would make the plugin less reusable. If the authors of pslist would have opted for print, we wouldn't be easily able to reuse their output. Generators use the `yield` statement instead of return. In our example it looks like this:

```
for process in processes:
    yield(0, [int(process['PID']),process['name'],bool(process['pslist']),bool(process['psscan'])])
```

The last change you need to make is to remove or comment out the exit and print instructions in lines 32 and 33. It should look like this:

```
#print("It's your job to finish that plugin --> \n\n")
#exit(1)
```

You can now go back to the terminal and try it out:

```
vol3 -f c2.vmem minipsxview
```