July, 2020

Windows Memory Forensics III

Investigating Process Objects and Network Activity.

LIFARS
your digital world, **secured**

Contents

244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

INVESTIGATING PROCESS OBJECTS

Generally speaking, an object is a data structure that represents a system resource, such as a file, thread, or graphic image. Kernel objects in Windows include thirty-seven different variants, including processes, threads, mutexes, files, registry hives etc. To access an object, an application must obtain a so-called object handle, which can be used to interact with the object. Each kernel object maintains information about a number of handles that are opened to it from user space, as well as the count of pointers to that object, used by kernel modules. Handle count is used to ensure that kernel object will not be destroyed if there still is some opened handle pointing to it.
By examining object handles we may be able to identify hidden processes. If any kernel object references some process in its handle table, it means that the process must exist, or must have been present on the system at some point in time.
Handles also reveal what objects were accessed by the process. For example, what files were manipulated, what registry keys have been accessed, etc.

HANDLES

Volatility plugin handles enumerates all handles that are opened on the investigated system. Keep in mind that output from this plugin is very long, as the number of opened handles per process can reach thousands. It is advised to limit output to handles belonging to a specified process (use command line argument *-p PID* or *-n partOfProcessName* to display handles of processes matching partial name), or to include only handles of specific type. The five most useful handle types for forensic purposes are:

- File for file objects,
- Process for process handles – parent process always has handle to spawned child process,
- Key for registry keys,
- Thread to see which threads belong to the process,
- Mutant to display mutex structures.

To limit a handles plugin output to specified types, run it with *-t <Type1,Type2,…>*. Note that types must be written case-sensitively. To further shorten the plugin output, use *-s* parameter which excludes unnamed handles. Most of the time it does not have any impact on the investigation.

Let us examine how we can leverage handles in an investigation. In previous parts of this series, we worked with an image from the system infected by malicious VisualBasic script. WScript.exe was invoked from command line to run the script named Judgement_04212020_2313.vbs.
We can look which process has a handle opened to this file (if any): run handles plugin, filter output to include only handles of type "File" and redirect output to GREP in order to find the file in question:

```
$vol.py -f /mnt/hgfs/AnalystVMShare/vbs-memdump.mem --profile Win8SP1x64 handles -t File |
grep -i judgement
Volatility Foundation Volatility Framework 2.6
[Note: column headers added, as GREP filtering cuts them out😊]
Offset(V)             Pid             Handle             Access Type             Details
------------------ ------ ------------------ ------------------ ---------------- -------

0xffffe0007b770570   1952             0x1224           0x100081 File
\Device\HarddiskVolume2\Users\Analyst\Desktop\!vzorky\Judgement_04212020_2313_2
0xffffe0007b577a40   1952             0x1238           0x100081 File
\Device\HarddiskVolume2\Users\Analyst\Desktop\!vzorky\Judgement_04212020_2313_2\Judgement_0421
2020_2313
0xffffe0007ac168e0   1952             0x1248           0x100081 File
\Device\HarddiskVolume2\Users\Analyst\Desktop\!vzorky\Judgement_04212020_2313_2\Judgement_0421
2020_2313
0xffffe0007b2f6570   1952             0x1254           0x100081 File
\Device\HarddiskVolume2\Users\Analyst\Desktop\!vzorky\Judgement_04212020_2313_2
```

Handles plugin helped us to identify four handles to Judgement-related files. However, they only refer to parent directories of the same name, not to the VisualBasic script itself. All handles were owned by process with PID 1925. We can consult pstree plugin output to verify which process the PID belongs to:

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/vbs-memdump.mem --profile Win8SP1x64 pstree
<snip>
0xffffe0007b486080:explorer.exe                      1952   1932     82      0 2020-05-18
14:04:02 UTC+0000
. 0xffffe0007b34a080:cmd.exe                         2592   1952      1      0 2020-05-18
14:06:43 UTC+0000
.. 0xffffe0007b4ca080:conhost.exe                    2608   2592      2      0 2020-05-18
14:06:43 UTC+0000
.. 0xffffe00079d9a900:wscript.exe                    1012   2592      4      0 2020-05-18
14:23:55 UTC+0000
<snip>
```

Based on the output, wscript.exe did not have a file handle itself. However, its parent's parent process, explorer.exe, did.

Handles become even more exciting when it comes to the winevt service. Why? The winevt service is responsible for updating Windows Event Log files, prominent sources of forensically relevant information. Because of that, its respective process (which is, by the way, one of svchost.exe instances running on any system) must have handles opened to any Event Log file, distinguishable by the .evtx extension:

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 handles -t File |
grep evtx
-----------
0xffffe0001bcd6c10    832              0x154           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Windows
Defender%4Operational.evtx
0xffffe0001d2a5670    832              0x1a8           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\System.evtx
0xffffe0001d3130f0    832              0x1ac           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Application.evtx
0xffffe0001d30e680    832              0x1b4           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-WorkFolders%4WHC.evtx
0xffffe0001d30a070    832              0x1bc           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Internet Explorer.evtx
0xffffe0001d30e230    832              0x1c0           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Security.evtx
0xffffe0001d309c70    832              0x1c4           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Windows
Defender%4WHC.evtx
0xffffe0001d30d9b0    832              0x1c8           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Windows PowerShell.evtx
0xffffe0001d30db10    832              0x1cc           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Key Management Service.evtx
0xffffe0001bcd6520    832              0x1d0           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Kernel-Power%4Thermal-
Operational.evtx
0xffffe0001d30cf20    832              0x1d4           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\HardwareEvents.evtx
0xffffe0001bd12f20    832              0x1d8           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Ntfs%4WHC.evtx
0xffffe0001bd11640    832              0x1dc           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Kernel-
Boot%4Operational.evtx
0xffffe0001d307480    832              0x1e0           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Kernel-
ShimEngine%4Operational.evtx
0xffffe0001d305400    832              0x1e4           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Ntfs%4Operational.evtx
0xffffe0001bd132f0    832              0x1f4           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
Wcmsvc%4Operational.evtx
0xffffe0001d2fe070    832              0x228           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Kernel-
WHEA%4Errors.evtx
0xffffe0001d2ff850    832              0x22c           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Kernel-
WHEA%4Operational.evtx
0xffffe0001b9e89e0    832              0x2f0           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Diagnosis-
DPS%4Operational.evtx
0xffffe0001b94e3f0    832              0x514           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
AppXDeploymentServer%4Operational.evtx
0xffffe0001b9512a0    832              0x524           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-TerminalServices-
LocalSessionManager%4Admin.evtx
0xffffe0001b9533c0    832              0x528           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-User Profile
Service%4Operational.evtx
0xffffe0001b953f20    832              0x52c           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-TerminalServices-
LocalSessionManager%4Operational.evtx
0xffffe0001b9637f0    832              0x554           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
NetworkProfile%4Operational.evtx
0xffffe0001b980b40    832              0x588           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Kernel-
StoreMgr%4Operational.evtx
0xffffe0001b99aa00    832              0x5d8           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
DeviceSetupManager%4Operational.evtx
0xffffe0001bcb9d90    832              0x610           0x12019f File
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
GroupPolicy%4Operational.evtx
0xffffe0001b9ba680    832              0x614           0x12019f File
```

FILESCAN

Plugin handles allows us to determine which file is accessed by a particular process. The presence of a handle is a clear indicator. But what if the handle has already been released?

Filescan is another "scanning" plugin that goes through the entire memory image, just like psscan did. Instead of looking up process objects, filescan searches for objects of file type. Even if there is no handle left for the file, it still can be present in the memory. Therefore, plugin filescan enables investigators to gain a deeper insight on what files have been present in memory, meaning that they have been manipulated somewhere in recent past. In contrast to handles, by scanning files we cannot determine which process opened the file. We only have information that a file has been opened. Like handles plugin, output produced by filescan can be long and therefore it is advised to redirect it to a file.

Look at the filescan output below. Highlighted is the Application.evtx file, one of the files Event Logs identified already with handles plugin. Filescan output shows the physical offset of the process object, which can be subsequently used with dumpfiles plugin to obtain a copy of the Application Event Log and analyze it, almost as if we have copied it out from the running system.

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 filescan
Volatility Foundation Volatility Framework 2.6
Offset(P)           #Ptr   #Hnd Access Name
------------------ ------ ------ ------ ----
0x0000000000907a50      1      0 R--r-d
\Device\HarddiskVolume2\Windows\System32\Tasks\Microsoft\Windows\AppID\VerifiedPublisherCertSt
oreCheck
0x0000000000907ba0     32      0 RW-rwd \Device\HarddiskVolume2\:$I30:$INDEX_ALLOCATION
0x0000000001066560     17      0 RW-rwd \Device\HarddiskVolume2\$Directory
0x00000000010ff7e0  32758      1 R----- \Device\HarddiskVolume2\System Volume
Information\{17b3dc75-9902-11ea-825d-080027ef1a8e}{3808876b-c176-4e48-b7ae-04046e6cc752}
0x00000000010ffe10     19      0 RW-rwd \Device\HarddiskVolume2\$Directory
<snip>
0x00000001a47130f0  32767      1 RW-r--
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Application.evtx
<snip>
```

DUMPFILES

Dumpfiles plugin allows us to (surprisingly 😊) dump files present in a memory image. Of course, we probably do not plan to dump every single file from memory. Dumpfiles plugin supports different ways of telling which file(s) we want to obtain. "By default, dumpfiles iterates through the VAD and extracts all files that are mapped as DataSectionObject, ImageSectionObject or SharedCacheMap," says Volatility documentation. Based on that, after running dumpfiles, three types of files can be obtained:

- img – ImageSectionObject
- dat - DataSectionObject
- vacb – SharedCacheMap

We can dump all files loaded in memory of a process, by defining a virtual offset to that process. Virtual offset is included in pslist or pstree plugins.

We can use *-r [regular_expression]* method to dump all files matching a defined regular expression. For example, to dump Security.evtx – Security event log – invoke Volatility with:

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 dumpfiles -r
Security.evtx -D /mnt/hgfs/AnalystVMShare/pony/evtx

Volatility Foundation Volatility Framework 2.6
DataSectionObject 0xffffe0001d30e230   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Security.evtx
SharedCacheMap 0xffffe0001d30e230   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Security.evtx
DataSectionObject 0xffffe0001b9aab50   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
SMBServer%4Security.evtx
SharedCacheMap 0xffffe0001b9aab50   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
SMBServer%4Security.evtx
DataSectionObject 0xffffe0001d233490   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Windows Firewall With
Advanced Security%4ConnectionSecurity.evtx
SharedCacheMap 0xffffe0001d233490   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-Windows Firewall With
Advanced Security%4ConnectionSecurity.evtx
DataSectionObject 0xffffe0001d9efbb0   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
SmbClient%4Security.evtx
SharedCacheMap 0xffffe0001d9efbb0   832
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Microsoft-Windows-
SmbClient%4Security.evtx
```

We always need to define the existing output directory to store extracted objects. The content is a bit messy, though. General naming convention makes up a dump named by the PID of the owning process (meaning that the file object has been found in memory location loaded by that process), combined with virtual offset of that file – as you can see for some files in Figure 1 below. We can use *-n* or *--name* to preserve the original filename in the dump. It is also possible to generate a summary file with *-s* option. A file with a list of dumped files will be generated.

However, not every file will be present in VAD, or active. Such files cannot be dumped using regex matching – they will be not found. In that case, we need to provide a physical offset of a file object present in memory, and that precious physical offset is provided by filescan plugin:

```
$ grep Application.evtx /mnt/hgfs/AnalystVMShare/pony/filescan.txt
0x00000001a47130f0  32767      1 RW-r--
\Device\HarddiskVolume2\Windows\System32\winevt\Logs\Application.evtx

$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 dumpfiles -n -Q
0x00000001a47130f0 -D /mnt/hgfs/AnalystVMShare/pony/evtx
```

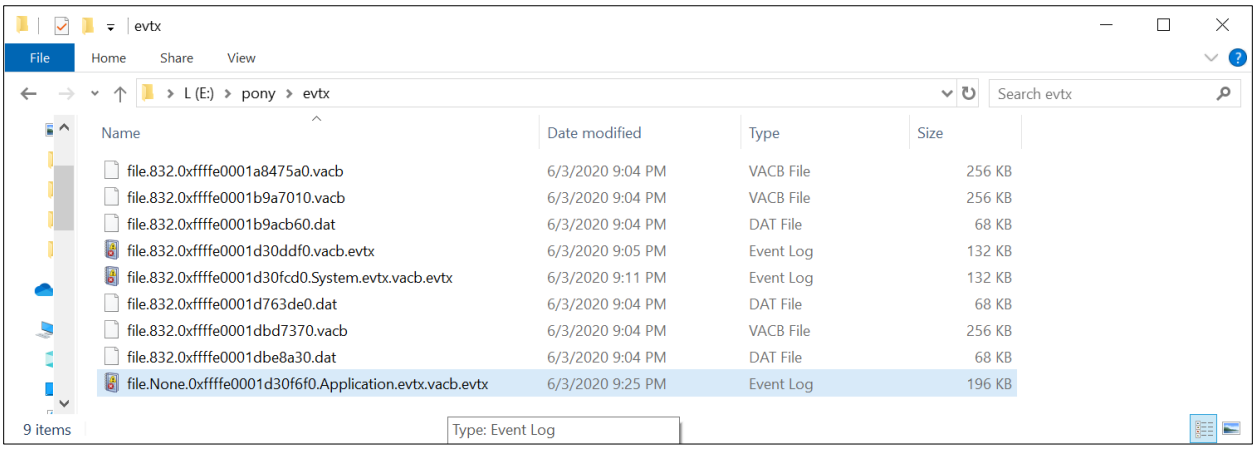
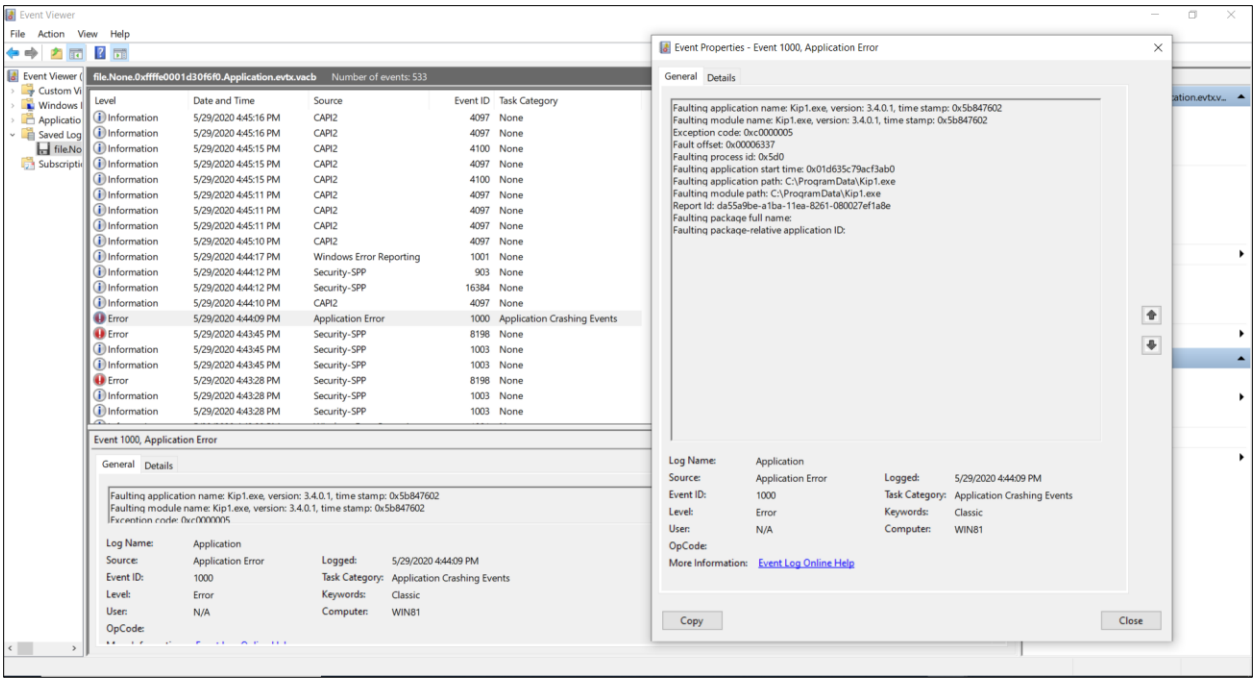


*Figure 1: Directory with dumped *Security.evtx*, System.evtx and Application.evtx.*

In the output above, note different extensions, added to dumped files. Files with evtx extension have already been renamed by an analyst.

If we are lucky enough, we may be able to open the extracted evtx file with no effort except adding ".evtx" extension. There may be problems with files being not fully contained in memory, and in that situation, we will need to attempt repairing files, or analyze them in a more robust or simpler way, depending on the type of data we are dealing with.

In this case, however, extracted Application log could have been analyzed directly. Only a few lines scrolled, and we were able to find the evidence of execution – and crashing – of kip1.exe malware component.



MUTANTSCAN

Mutantscan is a scanning plugin that goes through a memory image and searches for mutex kernel objects. Either this or handles plugin can reveal mutex structure that is often used by malware to ensure that once the system is infected, it will not be reinfected by the same malware. Mutant objects used by malware often have a unique or distinguishable name and thus can be used as an indicator of compromise.

ONE STEP FURTHER: EXECUTABLE EXTRACTION AND BRIEF ANALYSIS.

Let us show one more example about how we can leverage filescan and dumpfiles plugins and introduce new plugins – procdump and memdump.

So far, we can be pretty suspicious about process Kip1.exe. We can take a closer look at its executable file: first, we attempt to look for files of similar name in filescan plugin output:

```
$ grep -i kip1 /mnt/hgfs/AnalystVMShare/pony/filescan.txt
0x000000019e3d4360     18      0 R--r-d \Device\HarddiskVolume2\ProgramData\Kip1.exe
0x00000001a4767070      3      0 -W----
\Device\HarddiskVolume2\Users\Analyst\AppData\Local\Microsoft\Windows\WER\ReportArchive\AppCra
sh_Kip1.exe_fcdf1afa36b1c54fc31ae28ba1326995cdc11b84_7f7170f9_0a0237ee\Report.wer
0x00000001ab3e1da0      3      0 -W----
\Device\HarddiskVolume2\Users\Analyst\AppData\Local\Microsoft\Windows\WER\ReportArchive\AppCra
sh_Kip1.exe_fcdf1afa36b1c54fc31ae28ba1326995cdc11b84_7f7170f9_0129899d\Report.wer
0x00000002159c9e80     17      0 -W----
\Device\HarddiskVolume2\Users\Analyst\AppData\Local\Microsoft\Windows\WER\ReportArchive\AppCra
sh_Kip1.exe_94e939def2be8adb3fc71bf371a2c64a925cd024_6869bfc2_046b43a1\Report.wer
0x00000002190177c0     15      0 -W----
\Device\HarddiskVolume2\Users\Analyst\AppData\Local\Microsoft\Windows\WER\ReportArchive\AppCra
sh_Kip1.exe_fcdf1afa36b1c54fc31ae28ba1326995cdc11b84_7f7170f9_09d862b1\Report.wer
```

Yeah, it seems we have a candidate in ProgramData directory. If we recall what pstree plugin says about Kip1.exe process command line, it has been this very file:

```
0xffffe0001ab2e080:Kip1.exe                          1552   1724      4      0 2020-05-29
14:45:08 UTC+0000
    audit: \Device\HarddiskVolume2\ProgramData\Kip1.exe
    cmd: C:\ProgramData\Kip1.exe
    path: C:\ProgramData\Kip1.exe
```

We can dump the file, using the physical offset from filescan output:

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 dumpfiles -n -Q
0x000000019e3d4360 -D whitepaper/pony
Volatility Foundation Volatility Framework 2.6
ImageSectionObject 0x19e3d4360   None   \Device\HarddiskVolume2\ProgramData\Kip1.exe
DataSectionObject 0x19e3d4360   None   \Device\HarddiskVolume2\ProgramData\Kip1.exe

ls -lah whitepaper/pony
total 32M
...
-rw-rw-r-- 1 analyst analyst 768K Jun  4 08:35 file.None.0xffffe0001bc51010.Kip1.exe.img
-rw-rw-r-- 1 analyst analyst 768K Jun  4 08:35 file.None.0xffffe0001d7d40a0.Kip1.exe.dat
```

Verify this time, besides DataSection object we have dumped "img" file – ImageSection Object, so the process executable.

DUMPFILES VS. PROCDUMP VS. MEMDUMP

Is there any other way we can dump a process executable? Absolutely, procdump plugin is designed directly for this purpose. It operates on the process with the provided PID *(-p*), name (*-n*), or on the process object at the specified physical offset (*-o*).

Knowing the PID of Kip1.exe process, it is easy to obtain its copy:

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 procdump -p 1552
-D whitepaper/pony
Volatility Foundation Volatility Framework 2.6
************************************************************************
```

```
Writing Kip1.exe [  1552] to 1552.dmp
```

Is there any difference between files dumped by procdump and dumpfiles? An easy way to check is to look at file sizes. As we can see, the sizes are not lose in size:

```
$ ls -lah .
-rw-rw-r-- 1 analyst analyst 85K Jun  4 14:28 executable.1552.exe
-rw-rw-r-- 1 analyst analyst 768K Jun  4 08:35 file.None.0xffffe0001bc51010.Kip1.exe.img
```

These plugins work in very different ways: dumpfiles dumps the file object as it was loaded into memory. Procdump focuses on the process executable itself. To make things a bit more complicated, plugin memdump can be used to dump every memory section corresponding to a process defined by a user. It can include loaded DLLs, file objects, network endpoint objects – anything that was loaded in the process's memory. Running this plugin is very similar to creating a process dump on a live system, for example with SysInternals ProcDump.exe, or by rightclicking a process in Task Manager and choosing "Create a dump file" option.

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 memdump -p 1552 -
D whitepaper/pony
Volatility Foundation Volatility Framework 2.6
************************************************************************
Writing Kip1.exe [  1552] to 1552.dmp
```

Take a look at the third, and logically the largest dump retrieved from Kip1.exe process:

```
$ ls -lah .
-rw-rw-r-- 1 analyst analyst 85K Jun  4 14:28 executable.1552.exe
-rw-rw-r-- 1 analyst analyst 768K Jun  4 08:35 file.None.0xffffe0001bc51010.Kip1.exe.img
-rw-rw-r-- 1 analyst analyst 524M Jun  4 15:14 1552.dmp
```

Kilobytes are replaced by megabytes of data loaded in process memory pages.

Either of the three previously mentioned plugin outputs can be easily scanned by any antimalware solution present on an analyst's machine.

See what ClamAV thinks about the health status of the extracted files from dumpfiles:

```
$ clamscan whitepaper/pony
whitepaper/pony/file.None.0xffffe0001bc51010.Kip1.exe.img: Win.Ransomware.Hermes-6877144-0
FOUND
whitepaper/pony/file.None.0xffffe0001d7d40a0.Kip1.exe.dat: Win.Ransomware.Hermes-6877144-0
FOUND
<output truncated>
----------- SCAN SUMMARY -----------
Known viruses: 7139803
Engine version: 0.99.4
Scanned directories: 1
Scanned files: 16
Infected files: 2
Data scanned: 1.95 MB
Data read: 31.55 MB (ratio 0.06:1)
Time: 68.033 sec (1 m 8 s)
```

We can use the hash of the extracted executable to check VirusTotal or any hash database of our choice to see if the file is known as malicious or upload the file itself to one of available online analysis sites – e.g. HybridAnalysis.com or AnyRun.

Small insights into what the executable is attempting can be given by good old strings. See excerpts of running standard ANSI and Unicode strings against executable itself, and against file dumped by dumpfiles plugin.

Multiple cryptocurrencies are mentioned, as well as encryption and passwords-related terms too. "SAMSUnG" is the title name on the window that appeared on the infected system when the malware was run. Information about version of that executable follows. In the dumped file, strings InternalName – dagdrmmen – OriginalFilename - dagdrmmen.exe indicate that Kip1.exe executable may be known as dagdrmmen.

Throughout analysis of the dumped executable and its memory sections, it is necessary to confirm capabilities of an executable. From a memory analysis perspective, capabilities of a process can be inferred by investigating its DLLs.

Little-endian Unicode strings from dumped Kip1.exe file:

```
~$ strings -e l file.None.0xffffe0001bc51010.Kip1.exe.img
SOFTWARsdf Location
MSdsfFO
SOFdsfed Tools\MSINFO
PAdfs
TIPOFDAY.TXT
Options
DisplayCurrentTip
Show Tips at Startup
Value
About
Udligger
VS_VERSION_INFO
VarFileInfo
Translation
StringFileInfo
040904B0
Comments
 SAMSUnG
CompanyName
 SAMSUnG
FileDescription
 SAMSUnG
LegalCopyright
 SAMSUnG
LegalTrademarks
 SAMSUnG
ProductName
 SAMSUnG
FileVersion
3.04.0001
ProductVersion
3.04.0001
InternalName
dagdrmmen
OriginalFilename
dagdrmmen.exe
DFGHSSDFGHFGHFGH
VCHDFHJFGHFGHDFGHDFGH
```

ANSI strings from procdump executable:

```
!This program cannot be run in DOS mode.
<snip>
regbot.php
bot_id=%s&x64=%d&is_admin=%d&IL=%d&os_version=%d
Content-Type:
application/x-www-form-urlencoded
multipart/form-data; boundary=
Content-Disposition: form-data;
0xd3adc0d3
--%s
%sname="zip_file"; filename="%s-%s.cab"
Content-Type: vnd.ms-cab-compressed
\Microsoft\Windows\CurrentVersion
C:\Program files\Internet Explorer\iexplore.exe
accounts
\Martin Prikryl\WinSCP 2\Sessions
FileZilla
```

```
recentservers
sitemanager
Ipswitch\WS_FTP\Sites\ws_ftp
iexplore.txt
\Microsoft\Internet Explorer\IntelliForms\Storage2
       "domain": "%s",
       "httpOnly": %s,
       "name": "%s",
       "path": "%s",
       "secure": %s,
       "value": "%s"
Web Data
logins
Software
Install Directory
webappsstore
formhistory
Path
D877F783D5D3EF8C
credentials
CABINET
Valve\Steam
config
loginusers
wallet.dat
<snip>
Ethereum
keystore
Electrum
Bytecoin
Namecoin
monero-project
wallet_path
```

ANSI strings from dumped Kip1.exe file:

```
$ strings file.None.0xffffe0001bc51010.Kip1.exe.img

!This program cannot be run in DOS mode.
Rich
.text
`.data
.rsrc
MSVBVM60.DLL
Orreries
VB5!6&*
dagdrmmen
Agavers8
Orreries
```

DLLLIST

Processes need to load multiple DLLs to be able to run on a system. DLLs, or dynamically linked libraries, are executable files with functions that are shared by multiple programs. Many programs and processes use the library, but only one copy of shared code really needs to be loaded into the OS memory. Some of the DLLs are part

of operating system, others are authored by third parties. When a program starts, it loads some DLLs based on its import address table. This table contains calls to functions stored in external files (DLLs). Such process is called Load-Time Dynamic Linking. Because the functions used are included in program's executable, it can be statically analyzed to get an idea of what the program is doing.
However, an executable can load a library at run time using API calls. This process is called Run-Time Dynamic Linking. Because it takes place at runtime, it is not listed in the original executable. This makes it the perfect method for attackers to hide true activity of their code.
If we found anomalies in process list in previous steps, we can then proceed with getting more information about what capabilities those suspicious processes have by looking at the libraries the process uses. General Windows libraries include:

- kernel32.dll - primary dynamic library for Windows' base functions: file creation, memory management etc.
- gdi32.dll – Graphic Device Interface functions, low-level drawing, text output, basic graphics
- user32.dll – user GUI interface, dialog boxes, windows, …
- comdlg32.dll – dialog boxes, Open/Save functions
- crypt32.dll - certificate and cryptographic messaging functions in the CryptoAPI
- ws2_32.dll – Winsock API, TCP/IP networking
- advapi32.dll – Windows Registry access, security calls
- netapi32.dll – network interfaces management
- msvcrt.dll, msvcp*.dll – MS Visual C and C++ library, where * is replaced by version number

To get the list of DLLs loaded by a process from memory, Volatility provides plugin dlllist. It walks the doubly-linked list of _LDR_DATA_TABLE_ENTRY structures which is pointed to by the PEB's InLoadOrderModuleList. DLLs are automatically added to this list when a process calls LoadLibrary and they are not removed until FreeLibrary is called and the reference count reaches zero. The load count column tells you if a DLL was statically loaded (i.e. as a result of being in the exe or another DLL's import table) or dynamically loaded.
The output can get long, so it is advised to list only DLLs belonging to the suspicious Kip1.exe process using *-p PID* switch. If the process was not located in doubly-linked list pointed to by PsActiveProcessHead as a result of rootkit actions (hiding in plain sight), we will need to use psscan to get a physical offset of the process of our interest and afterwards use dlllist *--offset=[offset]* option. The plugin will look at physical location and locates the PEB of the hidden process.

The output includes base of a DLL, its size, load count and path to its file. Base address can be subsequently used with **dlldump** plugin to dump an executable file. Dlldump also works with *-p [PID]* to dump all DLLs of a specified process, with *--offset=OFFSET* when the process is hidden and we have acquired its physical offset with psscan, and

with *-r [regex]* to dump only DLLs with names matching the regular expression. End of short interrupt – let's get back to DLLs loaded by suspicious Kip1.exe process.

Process Kip1.exe loads standard system libraries, including several cryptography- and networking-related libraries. The only module loaded from another location than System32 is the process executable itself. Note that one of favorite techniques of malware authors is to replace legitimate DLL with adversary one, place it in specific location and subvert Windows mechanism of loading DLLs in a way that adversary DLL will be first in loading order. Names of both files can be the same, but the location cannot. Make sure that you check the location from which the DLLs are loaded. Another thing to keep an eye on are 3rd party modules, placed in User's directories, such as AppData, or in various Temp folders. Such DLLs can be delivered by malware, as those locations allow for standard user write. It is definitely easier to drop adversary code there than into, for example, System32 directory.

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 dlllist -p 1552
Volatility Foundation Volatility Framework 2.6
************************************************************************
Kip1.exe pid:   1552
Command line : C:\ProgramData\Kip1.exe

Base                             Size          LoadCount LoadTime                       Path
------------------ ------------------ ------------------ ------------------------------ ----
0x0000000000400000            0x1a000             0xffff 2020-05-29 14:45:08 UTC+0000
C:\ProgramData\Kip1.exe
0x00007ffbf03e0000           0x1ac000             0xffff 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\ntdll.dll
0x0000000077010000            0x4b000             0xffff 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\wow64.dll
0x0000000076f90000            0x68000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\system32\wow64win.dll
0x0000000077000000             0x9000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\system32\wow64cpu.dll
0x0000000000400000            0x1a000             0xffff 2020-05-29 14:45:08 UTC+0000
C:\ProgramData\Kip1.exe
0x0000000077060000           0x16e000             0xffff 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\ntdll.dll
0x0000000074ec0000           0x140000             0xffff 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\KERNEL32.DLL
0x0000000073700000           0x1d6000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\WININET.dll
0x00000000730b0000           0x170000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\WinSxS\x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.9600.17415_none_dad8722c5
bcc2d8f\gdiplus.dll
0x0000000072fb0000            0x7e000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\DNSAPI.dll
0x0000000076ac0000           0x188000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\CRYPT32.dll
0x0000000074cc0000            0x50000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\WS2_32.dll
0x0000000073b20000           0x14a000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\urlmon.dll
0x0000000074530000            0x9f000             0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\WINHTTP.dll
0x0000000074b60000           0x153000                0x6 2020-05-29 14:45:08 UTC+0000
C:\Windows\SYSTEM32\USER32.dll
0x0000000076320000           0x10e000                0x6 2020-05-29 14:45:09 UTC+0000
C:\Windows\SYSTEM32\GDI32.dll
0x0000000074d80000            0x7c000                0x6 2020-05-29 14:45:09 UTC+0000
C:\Windows\SYSTEM32\CRYPTBASE.dll
0x00000000747f0000            0x54000                0x6 2020-05-29 14:45:09 UTC+0000
C:\Windows\SYSTEM32\bcryptPrimitives.dll
0x0000000074d50000            0x27000                0x6 2020-05-29 14:45:09 UTC+0000
C:\Windows\SYSTEM32\CRYPTSP.dll
0x00000000741b0000            0x30000                0x6 2020-05-29 14:45:09 UTC+0000
C:\Windows\system32\rsaenh.dll
0x0000000074190000            0x1e000                0x6 2020-05-29 14:45:09 UTC+0000
C:\Windows\SYSTEM32\bcrypt.dll
0x0000000072270000            0x49000                0x6 2020-05-29 14:45:09 UTC+0000
C:\Program Files (x86)\Internet Explorer\ieproxy.dll
0x00000000744a0000            0x8b000                0x6 2020-05-29 14:45:09 UTC+0000

<output truncated>
```

ENUMFUNC

There is one important thing to be aware of: by listing the DLLs, we do not get any information on which exact functions the program uses from all functions the DLL provides. That being said, only having a library with functions useful for adversary actions does not mean that process really intended to use them. Deeper analysis needs to be conducted, and Volatility provides a plugin for that.

To list functions imported and exported by modules (= executable files) running on the system, use plugin enumfunc. It lists imports (from import address table) and exports of every PE file (Portable Executable file, for example executables and DLLs) on the system. Output can be limited to only imports (-I), only exports (-E), or to distinguish between kernel i/e (-K) or process i/e (-P).

Below is an example of functions imported by Windows system processes. Processing of this plugin can take a long time and is resource intensive. Output of enumfunc can become massive and unfortunately, it is not possible to limit output to precise process using, for example, *-p PID* switch, known from other plugins. That is the reason why we include only a snippet of the output to give you an idea of plugin work.

```
vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 enumfunc -P -I
Volatility Foundation Volatility Framework 2.6

Process              Type       Module              Ordinal    Address              Name
smss.exe             Import     smss.exe            727        0x00007ffbf04437c0
ntdll.dll!RtlComputeCrc32
smss.exe             Import     smss.exe            1363       0x00007ffbf042dd00
ntdll.dll!RtlUpcaseUnicodeChar
smss.exe             Import     smss.exe            381        0x00007ffbf0470d60 ntdll.dll!NtOpenKey
smss.exe             Import     smss.exe            991        0x00007ffbf04400a0 ntdll.dll!RtlGetVersion
smss.exe             Import     smss.exe            235        0x00007ffbf0470d30 ntdll.dll!NtClose
smss.exe             Import     smss.exe            1472       0x00007ffbf0402430 ntdll.dll!TpAllocTimer
smss.exe             Import     smss.exe            1512       0x00007ffbf0452d30 ntdll.dll!TpSetTimer
smss.exe             Import     smss.exe            460        0x00007ffbf0470fa0
ntdll.dll!NtQuerySystemInformation
smss.exe             Import     smss.exe            672        0x00007ffbf040b620
ntdll.dll!RtlAllocateHeap
smss.exe             Import     smss.exe            920        0x00007ffbf0410060 ntdll.dll!RtlFreeHeap
smss.exe             Import     smss.exe            563        0x00007ffbf0471240 ntdll.dll!NtSetValueKey
smss.exe             Import     smss.exe            1342       0x00007ffbf044b7c0
ntdll.dll!RtlUnicodeStringToInteger
smss.exe             Import     smss.exe            925        0x00007ffbf0412030
ntdll.dll!RtlFreeUnicodeString
smss.exe             Import     smss.exe            1016       0x00007ffbf0405db0
ntdll.dll!RtlInitUnicodeStringEx
smss.exe             Import     smss.exe            378        0x00007ffbf0470f70 ntdll.dll!NtOpenFile
smss.exe             Import     smss.exe            301        0x00007ffbf0470cb0
ntdll.dll!NtDeviceIoControlFile
smss.exe             Import     smss.exe            465        0x00007ffbf0470db0
ntdll.dll!NtQueryValueKey
smss.exe             Import     smss.exe            1015       0x00007ffbf041d4f0
ntdll.dll!RtlInitUnicodeString
...
...
```

CRSCC process imports – this time from several DLLs:

```
csrss.exe            Import     csrss.exe            22         0x00007ffbed5c4c60
CSRSRV.dll!CsrServerInitialization
csrss.exe            Import     csrss.exe            26         0x00007ffbed5c892c
CSRSRV.dll!CsrUnhandledExceptionFilter
csrss.exe            Import     CSRSRV.dll           929        0x00007ffbf044bb20 ntdll.dll!RtlGetAce
csrss.exe            Import     CSRSRV.dll           2026       0x00007ffbf04631d0 ntdll.dll!_stricmp
csrss.exe            Import     CSRSRV.dll           2136       0x00007ffbf046cea0 ntdll.dll!swprintf_s
csrss.exe            Import     CSRSRV.dll           505        0x00007ffbf0471160
ntdll.dll!NtResumeThread
csrss.exe            Import     CSRSRV.dll           525        0x00007ffbf0470d20 ntdll.dll!NtSetEvent
csrss.exe            Import     CSRSRV.dll           274        0x00007ffbf0471740
ntdll.dll!NtCreateSymbolicLinkObject
csrss.exe            Import     CSRSRV.dll           678        0x00007ffbf040b170
ntdll.dll!RtlAnsiStringToUnicodeString
csrss.exe            Import     CSRSRV.dll           381        0x00007ffbf0470d60 ntdll.dll!NtOpenKey
...
csrss.exe            Import     basesrv.DLL          29         0x00007ffbed5c2b70
CSRSRV.dll!CsrValidateMessageBuffer
csrss.exe            Import     basesrv.DLL          28         0x00007ffbed5c1bf0
CSRSRV.dll!CsrUnlockThread
csrss.exe            Import     winsrv.DLL           295        0x00007ffbedb7cfe0
USER32.dll!GetClientRect
csrss.exe            Import     winsrv.DLL           488        0x00007ffbedb7cc50 USER32.dll!InflateRect
csrss.exe            Import     winsrv.DLL           609        0x00007ffbedb74200 USER32.dll!OffsetRect
csrss.exe            Import     winsrv.DLL           581        0x00007ffbedb7d2f0
USER32.dll!MapWindowPoints
```

NETWORK: WHO DO YOU TALK TO?

More than often, the very first indicator of adversary action in the infrastructure is some type of network monitoring alert. For example, your SIEM alerts on large volumes of data outgoing to a location outside your network, in other cases it is the EDR solution which spots a host communicating with a known C2 IP, or a malicious domain name is being resolved by internal DNS after one of the infrastructure servers issued such query. Network communication information is very volatile, but networking also "causes" creation of kernel objects of specific types, which are maintained in memory while the communication takes place (and possibly after it has ended, but objects have not been overwritten yet). Therefore, it is possible to get information about network endpoints and sockets using memory analysis.

NETSCAN

Netscan plugin, as the name suggests, goes through the memory image and searches for kernel objects related to networking: sockets, TCP and UDP endpoints, TCP listeners. All these objects have their unique pool tag that allows for searching these structures. To analyze Windows XP networking memory objects, four separate plugins are necessary: connections, connscan, sockets and sockscann. Due to changes in memory layout and objects in Vista+ operating systems, a new Volatility plugin was designed, encompassing functions of all the previous four: netscan.

By default, netscan output includes the following fields:

- Physical offset of the object
- Protocol used – TCPv4 or v6, UDPv4 or v6
- Local and foreign address
- State of connection – closed, established, listening, …
- PID of process that uses the connection
- Owner – process name, pointer to owning EPROCESS structure
- Time created – available only for UDP listeners

What should we watch out for while investigating network activities?

First, check which processes communicate on the network. Are these processes expected to be communicating? For example, most of system processes are not. So, if we spot paint.exe or wininit.exe chatting happily with some Internet IP address, raise a red flag. In general, any communicating process that is not a browser should be checked.

Next, check for any connection that may indicate lateral movement or data exfiltration. An excellent example would be an outbound connection to the local Windows filesharing ports – 135-139 (NetBios) and 445 (SMB). When discussing ports, there is another rule of thumb: connections from high ports (1024 and higher) to low ports shall be OK in the case of a workstation connecting to a server. Otherwise, a check is needed. High port to high port communication can be legitimate, but may very well be malicious. The investigator should check if such connection is caused by C2, or by some legitimate application used in the investigated infrastructure.

Does the netscan show anything peculiar on our malware/ransomware memory image?

244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

```
vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 netscan
Volatility Foundation Volatility Framework 2.6

Offset(P)          Proto    Local Address                  Foreign Address      State
Pid      Owner          Created
0x10bbf18d0        UDPv4    192.168.56.101:24577           *:*
4        System         2020-05-29 14:43:33 UTC+0000
0x10bbfb330        UDPv4    192.168.56.101:24577           *:*
4        System         2020-05-29 14:43:33 UTC+0000
0x10bba8ed0        TCPv4    0.0.0.0:445                    0.0.0.0:0            LISTENING
4        System
0x10bba8ed0        TCPv6    :::445                         :::0                 LISTENING
4        System
0x10ba426d0        TCPv4    10.0.2.15:1072                 173.239.5.6:80       CLOSE_WAIT
764      iexplore.exe
0x10ba76ba0        TCPv4    -:1278                         -:443                CLOSE_WAIT
764      iexplore.exe
0x10ba94780        TCPv4    10.0.2.15:1080                 52.218.104.106:443   CLOSE_WAIT
764      iexplore.exe
0x10baaa240        TCPv4    10.0.2.15:1103                 185.17.117.173:443   CLOSE_WAIT
764      iexplore.exe
0x10bab6900        TCPv4    10.0.2.15:1095                 185.17.117.173:443   CLOSE_WAIT
764      iexplore.exe
```

If we check all of the outbound IP addresses, listed by netscan plugin, and perform threat intel search, we will find out that highlighted **173.239.5.6:80** is indeed related to Kpot malware campaign. The process communicating with it is iexplore.exe, web browser, which is expected to be communicating over network. No other malicious IP addresses or suspicious connections were identified.

When using netscan plugin, unexpected results can occur. For example, PID of process can be invalid. This is due to the nature of scanning plugins – they can find objects that are not used anymore. If we find (for example) a TCP endpoint object whose connection is terminated, this object is just waiting for being overwritten. By following the owner – pointer to owning _EPROCESS structure – plugin tries to parse process structure at that location. But, in case this region has already been reused, parsing cannot lead to producing legitimate data, therefore PID is wrong and process name is probably not present.

In previous versions of Volatility, problems with parsing newer networking memory objects sometimes lead to displaying "-1" as PID when analyzing newer OS builds. When occurring this issue, there is still chance to learn about what the context of connection has been. Using Volatility's yarascan plugin, we can search for IP address of interest and look at surrounding data.

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 yarascan -Y
"173.239.5.6"

Volatility Foundation Volatility Framework 2.6
Rule: r1
Owner: Process iexplore.exe Pid 180
0x0c3edf3d  31 37 33 2e 32 33 39 2e 35 2e 36 3c 2f 52 49 3e   173.239.5.6</RI>
0x0c3edf4d  3c 48 49 50 3e 30 2e 30 2e 30 2e 30 3c 2f 48 49   <HIP>0.0.0.0</HI
0x0c3edf5d  50 3e 3c 55 49 3e 65 63 36 66 38 32 61 37 30 63   P><UI>ec6f82a70c
0x0c3edf6d  62 36 31 31 32 66 37 35 31 34 62 35 37 30 37 38   b6112f7514b57078
0x0c3edf7d  62 30 35 61 30 33 30 62 63 63 62 30 31 61 32 35   b05a030bccb01a25
0x0c3edf8d  35 39 66 65 39 33 35 66 64 61 35 32 64 63 66 36   59fe935fda52dcf6
0x0c3edf9d  64 35 37 37 63 39 3c 2f 55 49 3e 3c 53 3e 36 35   d577c9</UI><S>65
0x0c3edfad  3c 2f 53 3e 3c 44 49 3e 32 31 37 2e 32 33 2e 32   </S><DI>217.23.2
0x0c3edfbd  35 34 2e 31 32 34 3c 2f 44 49 3e 3c 59 3e 3c 54   54.124</DI><Y><T
0x0c3edfcd  3e 42 7c 30 7c 31 30 30 2e 30 30 30 30 3c 2f 54   >B|0|100.0000</T
0x0c3edfdd  3e 3c 54 3e 49 7c 30 7c 31 30 30 2e 30 30 30 30   ><T>I|0|100.0000
0x0c3edfed  3c 2f 54 3e 3c 54 3e 44 7c 30 7c 31 30 30 2e 30   </T><T>D|0|100.0
0x0c3edffd  30 30 30 3c 2f 54 3e 3c 54 3e 50 7c 30 7c 31 30   000</T><T>P|0|10
0x0c3ee00d  30 2e 30 30 30 30 7c 36 2e 35 30 30 30 3c 2f 54   0.0000|6.5000</T
0x0c3ee01d  3e 3c 54 3e 46 7c 32 7c 30 2e 30 30 37 30 7c 30   ><T>F|2|0.0070|0
0x0c3ee02d  2e 30 30 37 30 7c 30 2e 30 30 37 30 3c 2f 54 3e   .0070|0.0070</T>
```

CONCLUSION

In this part of *Windows Memory Forensics* series, we have covered a multitude of important topics. We focused on investigating process objects, such as handles, files, mutants. We have introduced plugins that can be used to dump objects and files from memory. We discussed the importance of DLLs in an investigation. Finally, we described the usage of netscan plugin and the basics of network communication review.