Plugins documentation:

Aviel Zohar

Introduction

This pdf contains documentation on 3 different files – volexp.py, pfn.py and winobj.py.

each of them contains multiple plugins (except winobj.py).

1. Vol3xp.py contains:
    - VolExp
    - StructAnalyzer
    - WinObjGUI
    - FileScanGUI

2. PFN.py contains:
    - P2V
    - PFNInfo
    - RAMMap

3. WinObj.py contains:
    - WinObj

VolExp.py

VolExp – Volatility Explorer

Introduction

This program allows the user to access a Memory Dump. It can also function as a plugin to the Volatility Framework (https://github.com/volatilityfoundation/volatility).

This program functions similarly to Process Explorer/Hacker, but additionally it allows the user access to a Memory Dump. This program can run from Windows, Linux and MacOS machines, but can only use Windows memory images.

Motivation

To this day, memory forensics analysts most frequently use CLI-based tools for memory analysis. For the sake of comfort, ease of navigation, and faster, more efficient analysis, I decided to create a tool that we're all familiar with from SysInternals [ProcExp/Process Hacker] as a GUI-based plugin for Volatility.
Additionally, I created it for my own enhanced understanding of the Volatility Framework.

This tool uses multithreading and multiprocessing to run different plugins (svcscan, hivelist, getsids, pslist, handles) and different calculations for automation and efficiency.

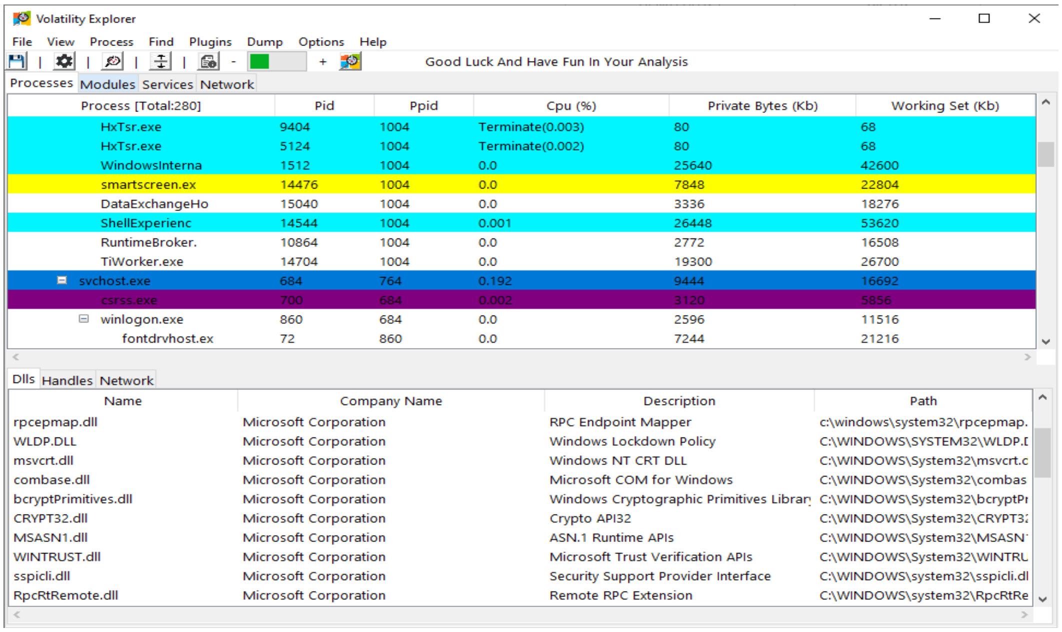
For the functionality of this plugin, I have written several additional plugins to the framework, very similar to volatility2 (Converted to volatility3 for compatibility) – getsids, getservicesids, privileges and envars and also added to the windows\symbols\extensions__init__.py file of the framework some functions to help me create these plugins.

Startup Arguments

This plugin accepts the following arguments:

- --DUMP-DIR: Directory to dump files.

- --SAVED-FILE: File that contains a previously saved run (you can save your progress at any time, similar to ida .idb file).
- --API-KEY: Virus Total API key to integrate with Virus Total.



Some Research

Firstly, I noticed a problem in the implementation of a lot PsTree tools (much more prevalent on later versions of Windows because there is a far greater number of processes).

Since many processes start and/or are terminated, a situation may occur where Process A creates process B as its child process, it is then terminated, and later process C uses process A's PID since it recognizes it as free – resulting in PsTree incorrectly presenting Process C as Process B's parent.

To fix the problem above, I looked for a kernel struct (so User Mode Malware can't trick us) that can give me solid proof of processes' parent-child relationships, and found it in _EPROCESS.CreateTime (so I can validate the parent process's creation time and that it is not at a later time than its child).

Secondly, I had to understand how to identify the process type (.Net, Immersive, Protected or Service), so I can visually categorize them using colors:

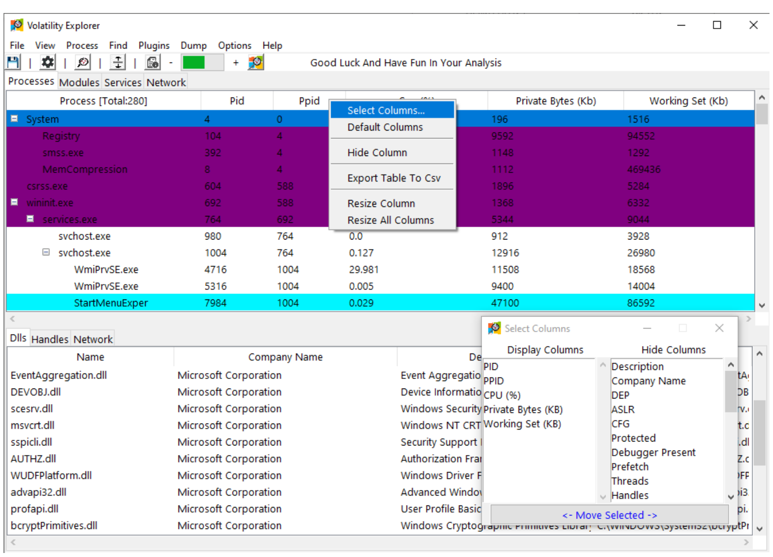
- Identifying a .Net Process (Yellow) - CrossSessionCreate and WriteWatch flags are on.
- Identifying an Immersive Process (Metro apps - Blue) – must be under a Job and (_EPROCESS.TokenFlags & token flag LOWBOX) or “S-1-15-2-“ is inside the token sids. This sid is reserved for Immersive processes.
- Identifying a Protected Process (Purple) – ProtectedProcess flag in Windows Vista to Windows 8.1; in later versions using the Protection struct inside the _EPROCESS.
- Services (Pink) – using the svcscan (plugin) output.

```
# Colored .Net processes. (os is vista or later)
if self.os_version['Major'] > 5 and e_proc.CrossSessionCreate == 1 and e_proc.WriteWatch == 1:
    process_comments[int(pid)] += "(Colored in yellow because this is a .Net process)"
    process_comments['pidColor'][int(pid)] = "yellow"

# Colored Immersive process. (os is vista or later)
TOKEN_LOWBOX = 0x4000 # this flag mean this is AppContainer!.
if proc_token and (self.os_version['Major'] > 5 and  e_proc.Job !=0 and proc_token.TokenFlags & TOKEN_LOWBOX or any("S-1-15-2-" in sid for sid in proc_token.get_sids())):
    process_comments[int(pid)] += "(Colored in turquoise because this is a Immersive process)"
    process_comments['pidColor'][int(pid)] = "turquoise1"
    integrity = 'AppContainer' if integrity not in integrity_levels else integrity

protection = 'Disable'
# Colored Protected process. (os is 8.1 or later)
if (self.os_version['Major'] == 10 or (self.os_version['Major'] == 6  and self.os_version['Minor'] == 3)) and e_proc.Protection.Type > 0:
    process_comments[int(pid)] += "(Colored in purple because this is a Protected process)"
    process_comments['pidColor'][int(pid)] = "purple"
    protection = 'Protected ()'.format(e_proc.Protection.Level) if int(e_proc.Protection.Level) not in protect_signer_by_level else protect_signer_by_level[int(e_proc.Protection.Level)]
elif self.os_version['Major'] > 5: # Check if vista or later
    protection = 'Protected' if int(get_right_member(e_proc, ["ProtectedProcess"]) or -1) == 1 else 'Disable'
```

Abilities

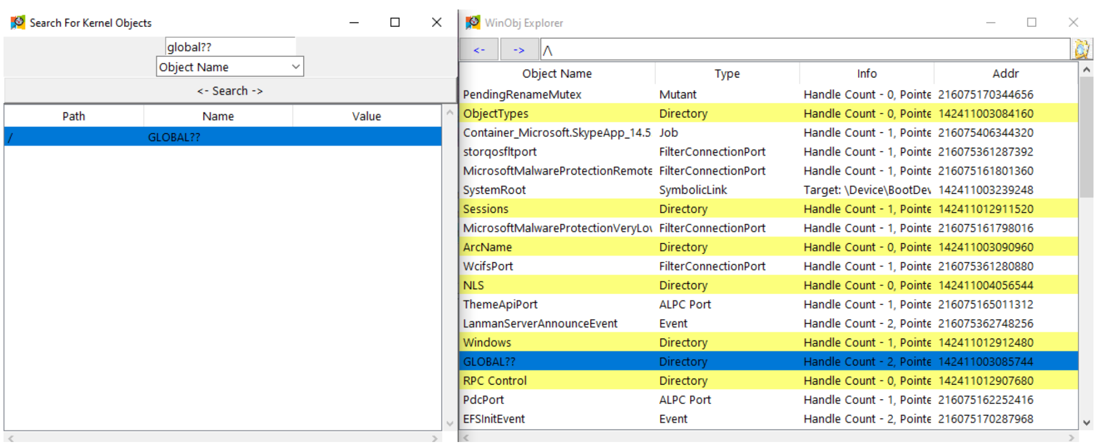
General UI

- The process tree hierarchy Table that is presented in the picture below has a functionality to add or remove columns from the table (using Right Click in the table Headers ->Select Columns, select some columns and click <-Move Selected-> button).
  You can also restore the table to the default state, hide selected columns, export the table as to a csv file, and resize specific columns/all of the table (you can also sort-by-column and change the order of the columns by dragging and dropping a column or by changing the order in the Select Column window).
  Additionally, you can type something while a row is selected to navigate to the row of the process typed (for example if I type “ser” in the main table below, the services.exe process will selected [or another process that starts with ser, if there are any]).
- The above is true for all the columns in the plugin, including modules, services, dlls, imports, etc.



- The main table also has a search bar (using ctrl+f/Find from the menu) that will search for handles/dlls.

- Options - > Change Themes to change the themes of the program (using tkinter themes, you can add more themes by installing the ttkthemes package [pip install ttkthemes]).
- Options -> Options[ctrl+o] to change configuration of the plugin: Add VirusTotal API key to integrate with VirusTotal, change dump directory, show unnamed handles, show PE string on PE properties, etc.
- By clicking on the progress bar, you are able to see what's currently running and what finished running, so you can track VolExp's current and historical progress (tasks that you run manually will also be added to the progress bar, like VAD information, PE information, etc.).
- You can also save your work to a file (File->Save/Save As) and then reload the plugin (using the –SAVED-FILE option) to restore from that specific point, to save all your work, the progress of the plugin, process comments and colors.
- Another very important UI class of this plugin, is the Explorer window.



This explorer is built upon the table (so everything you can do with the table you can do with the explorer as well).

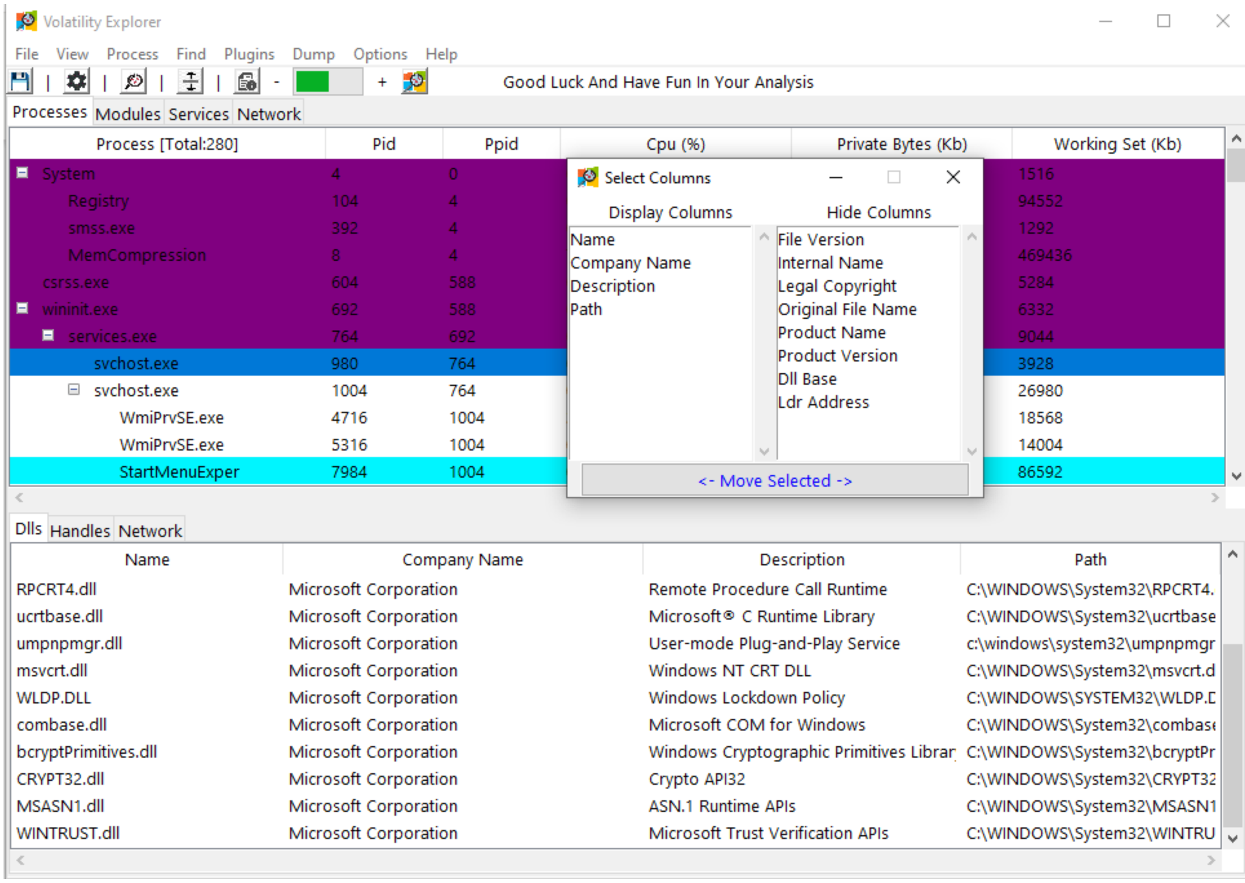
1. You can enter rows that are colored in yellow (much like folders in Windows File Explorer).

2. Export the entire Explorer to a csv and not just the presented table.

3. Using Ctrl + F to search for a specific string inside a specific column and navigate to that column (by double clicking the specific item).

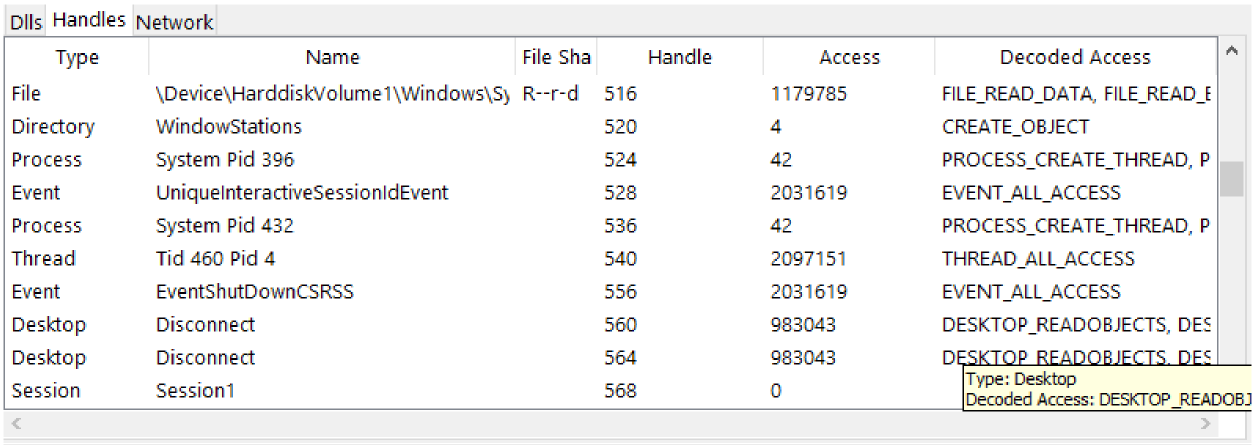
Functionality

- Process tree hierarchy with dlls, handles and connections to each process (there is also a tab for all the kernel modules, services and network connections).



As you can see, you can also see the Company Name and the Description of the dll by default (this will load while the plugin is running). You can also add additional colums such as Internal Name, Copyright, Original Name, Product Name and Product Version, etc.

- Mapping of the meaning behind handles' granted_access (the number representing a handle's permission) to a coherentive, literal meaning for each handle type.

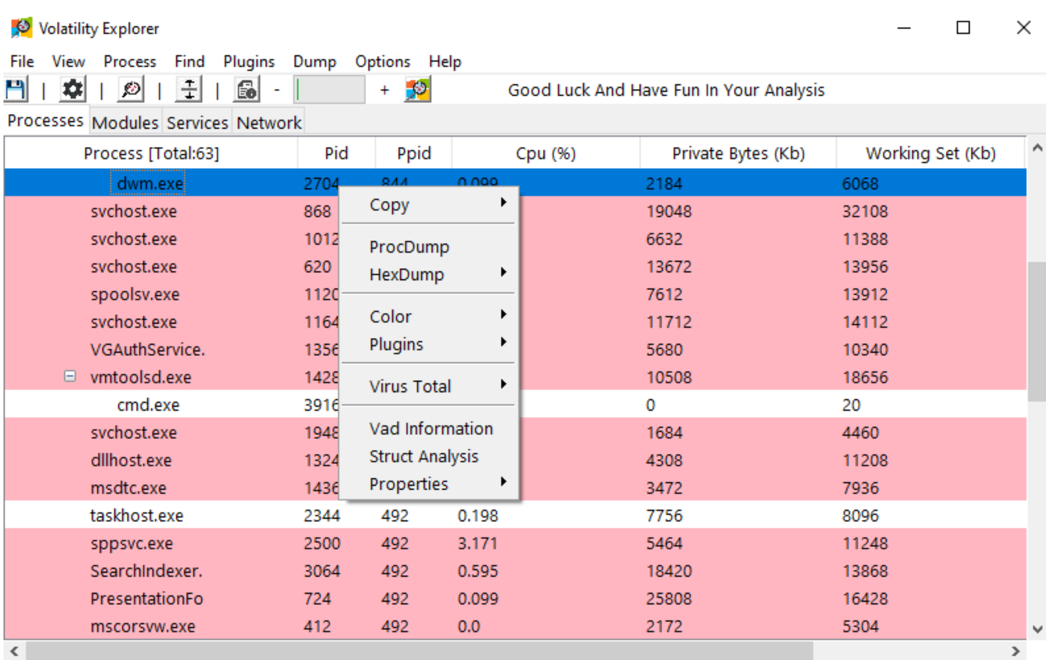


Dlls Handles Network

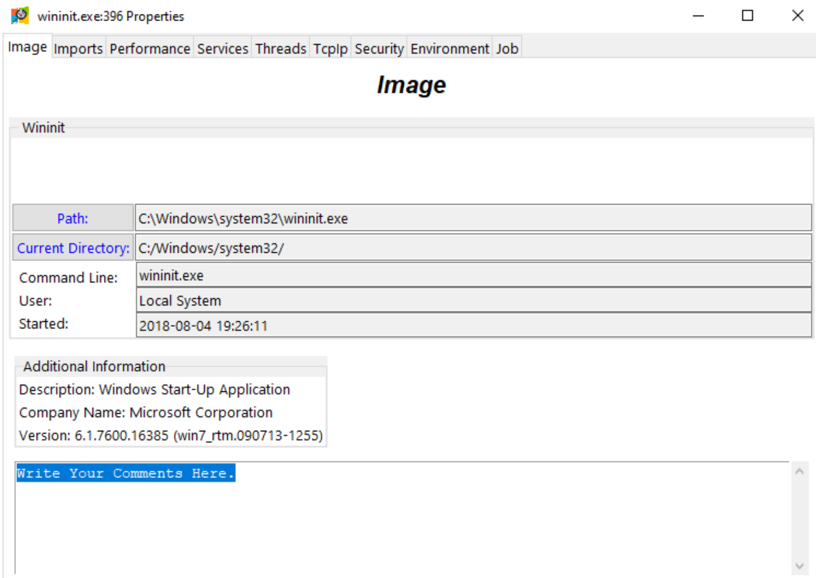
Type	Name	File Sha	Handle	Access	Decoded Access
File	\Device\HarddiskVolume1\Windows\Sy	R--r-d	516	1179785	FILE_READ_DATA, FILE_READ_E
Directory	WindowStations		520	4	CREATE_OBJECT
Process	System Pid 396		524	42	PROCESS_CREATE_THREAD, P
Event	UniqueInteractiveSessionIdEvent		528	2031619	EVENT_ALL_ACCESS
Process	System Pid 432		536	42	PROCESS_CREATE_THREAD, P
Thread	Tid 460 Pid 4		540	2097151	THREAD_ALL_ACCESS
Event	EventShutDownCSRSS		556	2031619	EVENT_ALL_ACCESS
Desktop	Disconnect		560	983043	DESKTOP_READOBJECTS, DES
Desktop	Disconnect		564	983043	DESKTOP_READOBJECTS, DES
Session	Session1		568	0	

Type: Desktop
Decoded Access: DESKTOP_READOBJ

- Process right click will let you view specific data related to this process.



For example, let's see the properties and Memory Information:

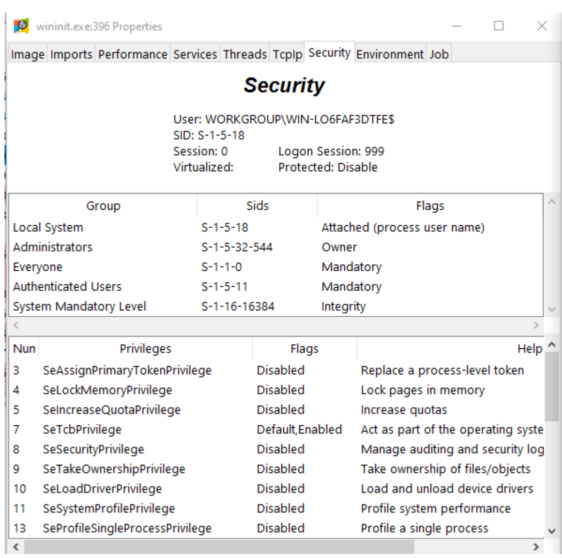


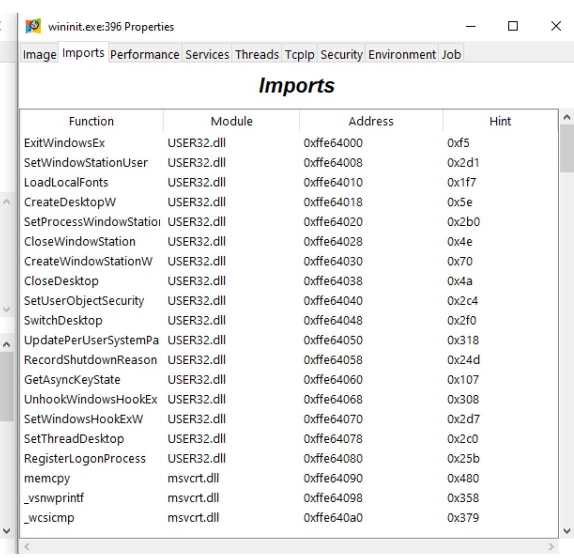
As you can see Process Properties contains a lot of tabs. In the first tab you can view general information.

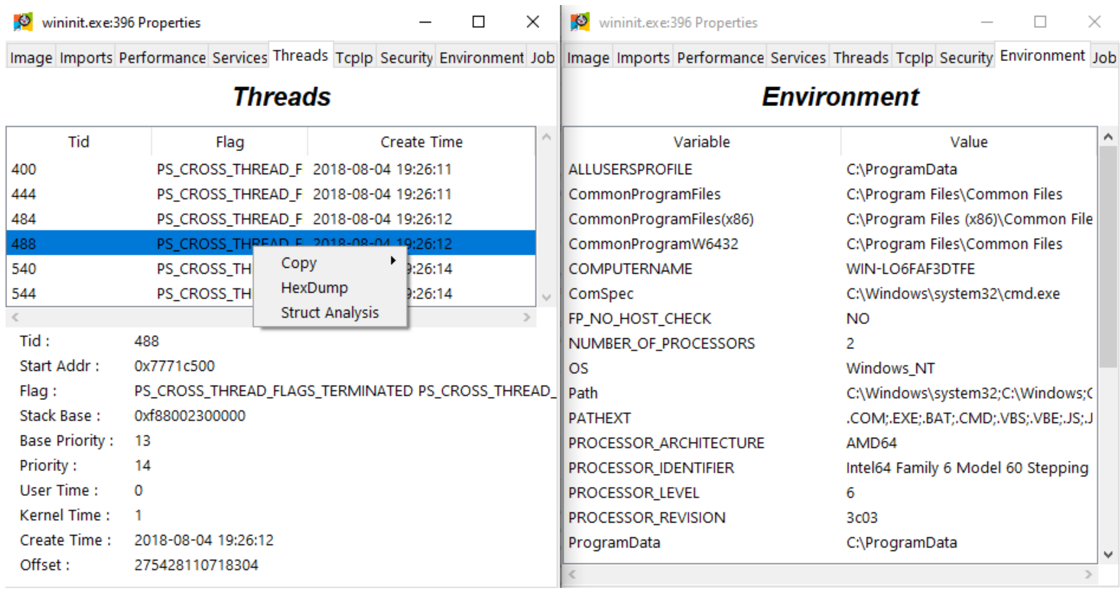
You can also add a comment to this specific process (as well as coloring processes [using right click on the selected process->Change Color].

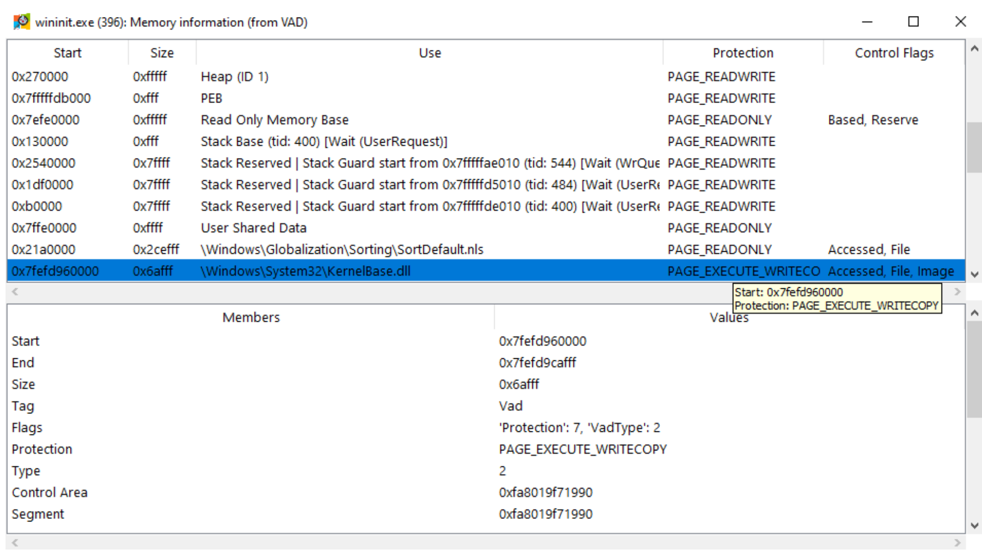
There is a lot more information in the other tabs, like the Process Imports, Performance, Services, etc. - Very similarly to Process Explorer.

Moving on to Security, Imports, threads and environment:



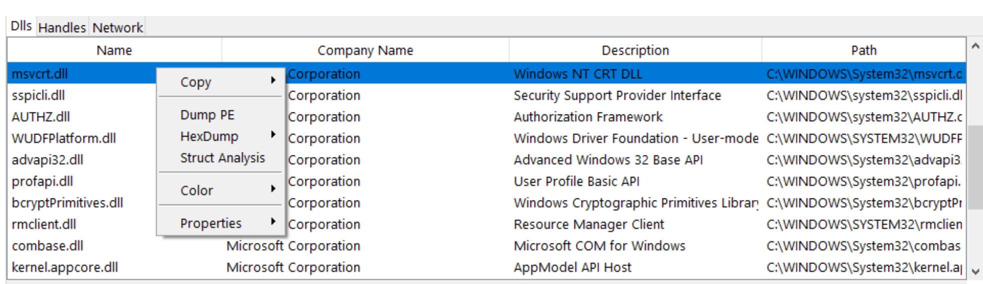




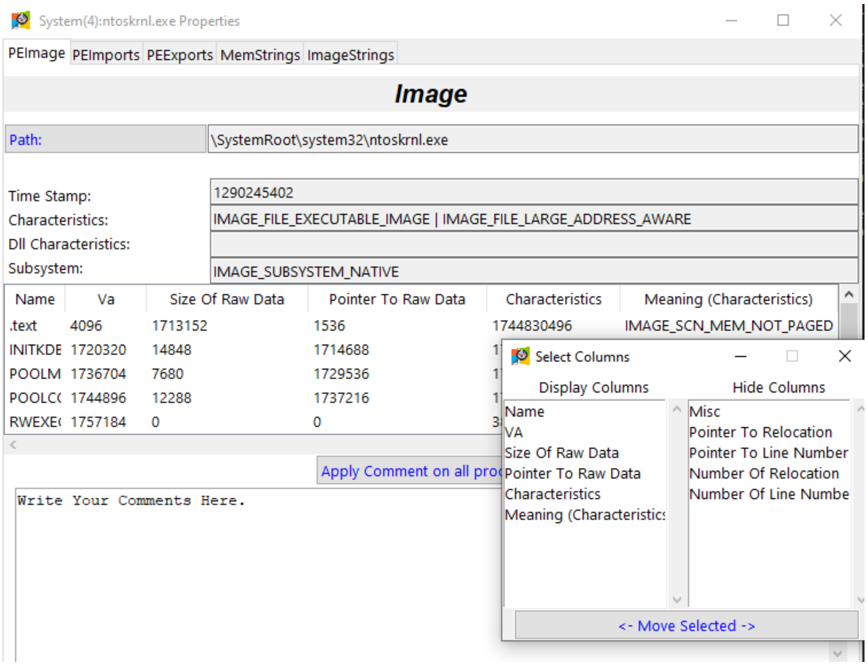


As you can see, memory information will display virtual address mapping information for a specific process, with a detailed description for each memory region.

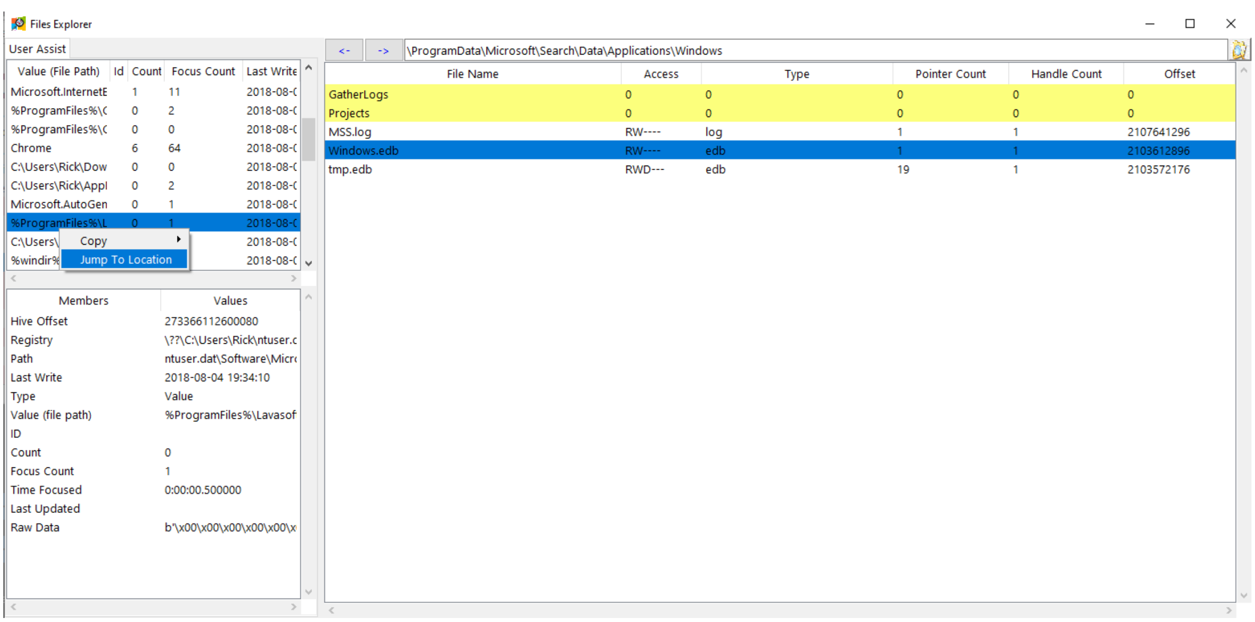
- PE File: You can also get information about PE files (from main table modules and from process dlls) by right clicking on a PE file:



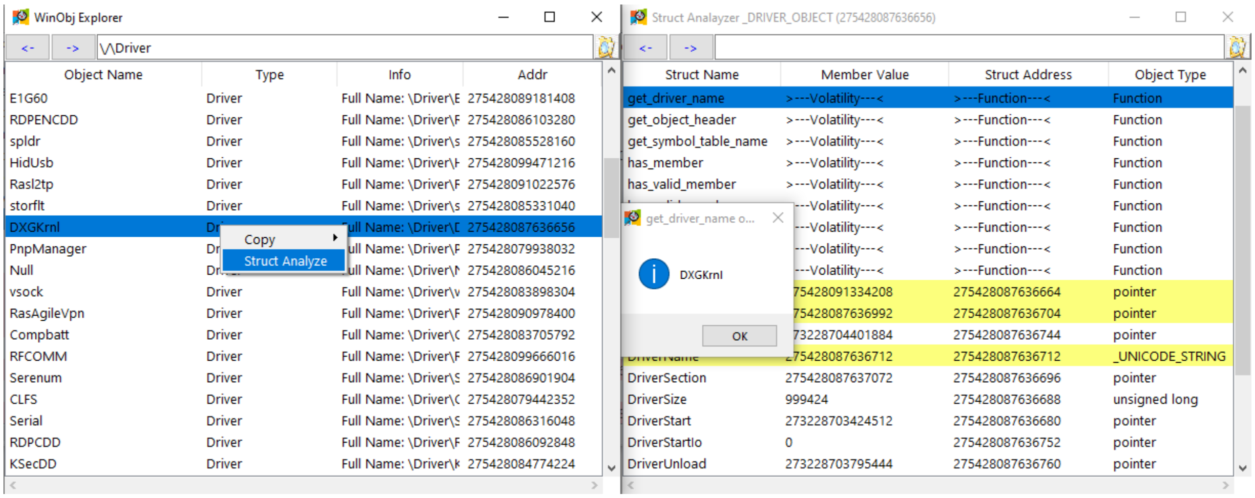
For example, clicking Properties will open the following window:



- Integrate with FileScanGui and WinObjGui: This plugin also gets information about files in memory (using FileScan), and Windows kernel objects (using my other plugin – WinObj below) to display them both in Explorer (MFT Parser GUI will be added as soon as Volatility will add this plugin, the code for this is already ready):



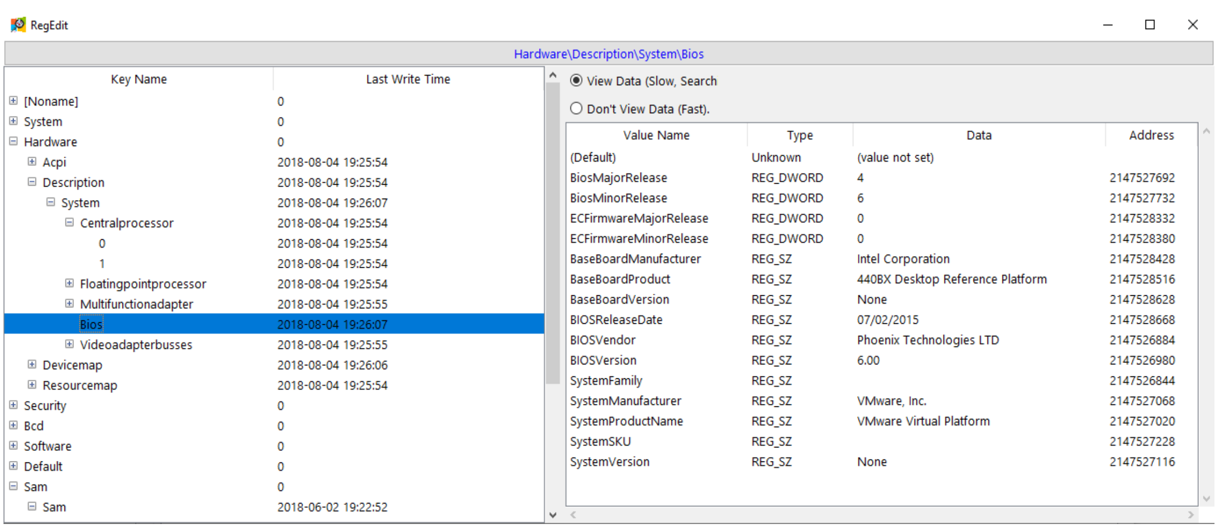
As you can see, the files are collected inside an Explorer UI with user assist data (using userassist plugin). You can jump directly to a specific file from the user assist.



Here you can see WinObj Explorer run Struct Analyze on a specific driver, and then run the volatility function get_driver_name on that driver which will result in the popup window DXGrnl.

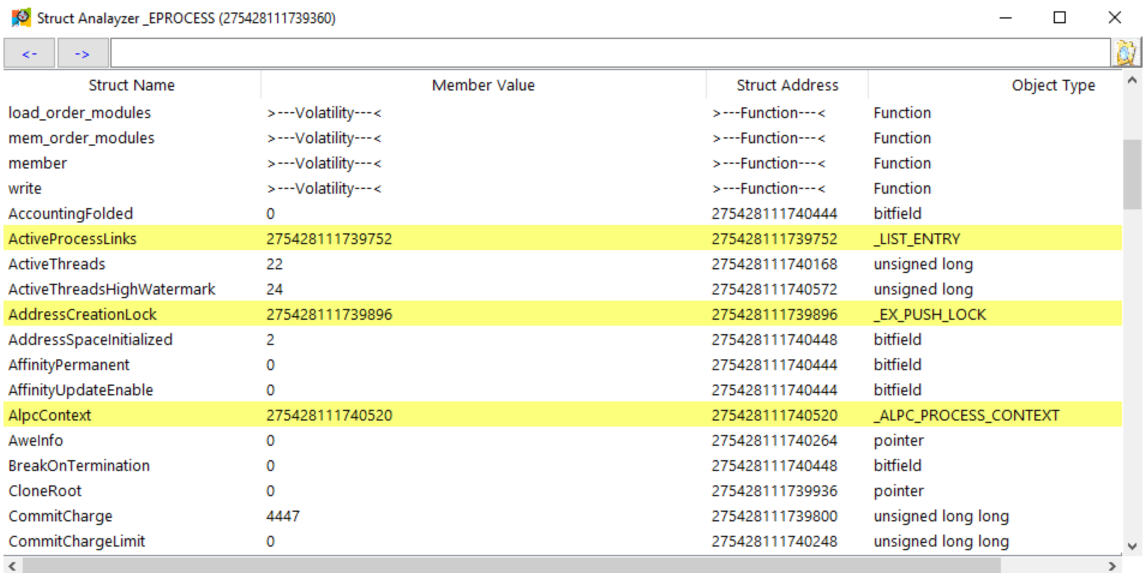
WinObj Explorer has [Struct Analyzer] on a lot of windows kernel objects, like _OBJECT_DIRECTORY, _DRIVER_OBJECT, _KMUTANT, etc.

- Registry Viewer – will display all the registry keys very similarly to Regedit, and will let you enumerate the values as well as the data.

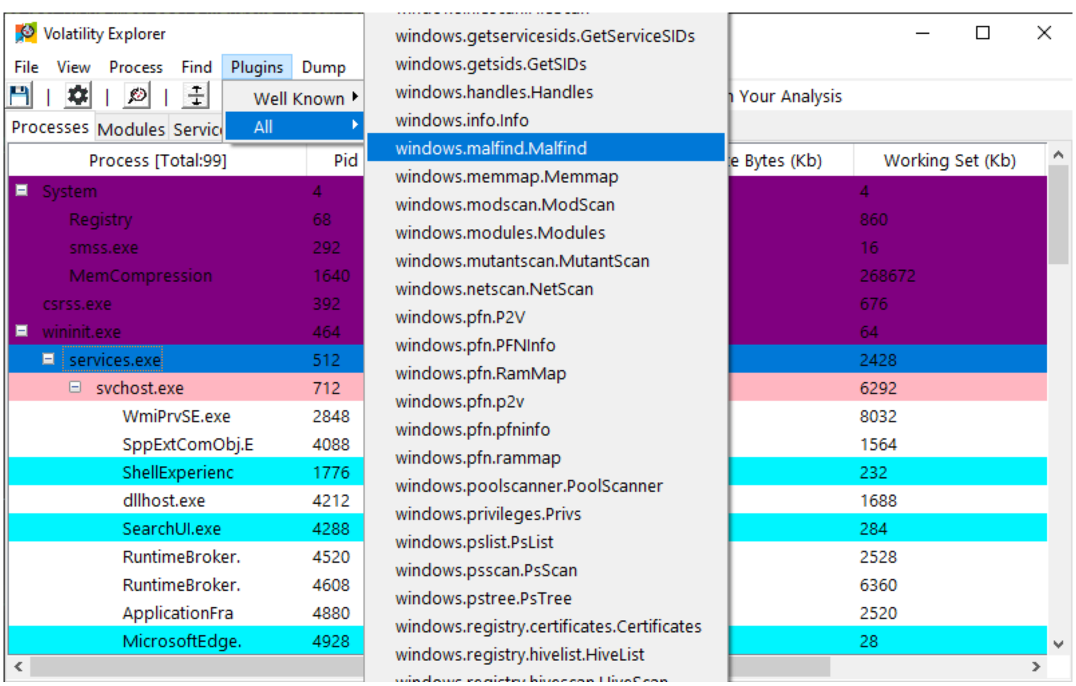


- Integrate With Struct Analyzer: For each process, PE file, thread, etc. you can right click -> Struct Analyzer to view much more information of each struct.

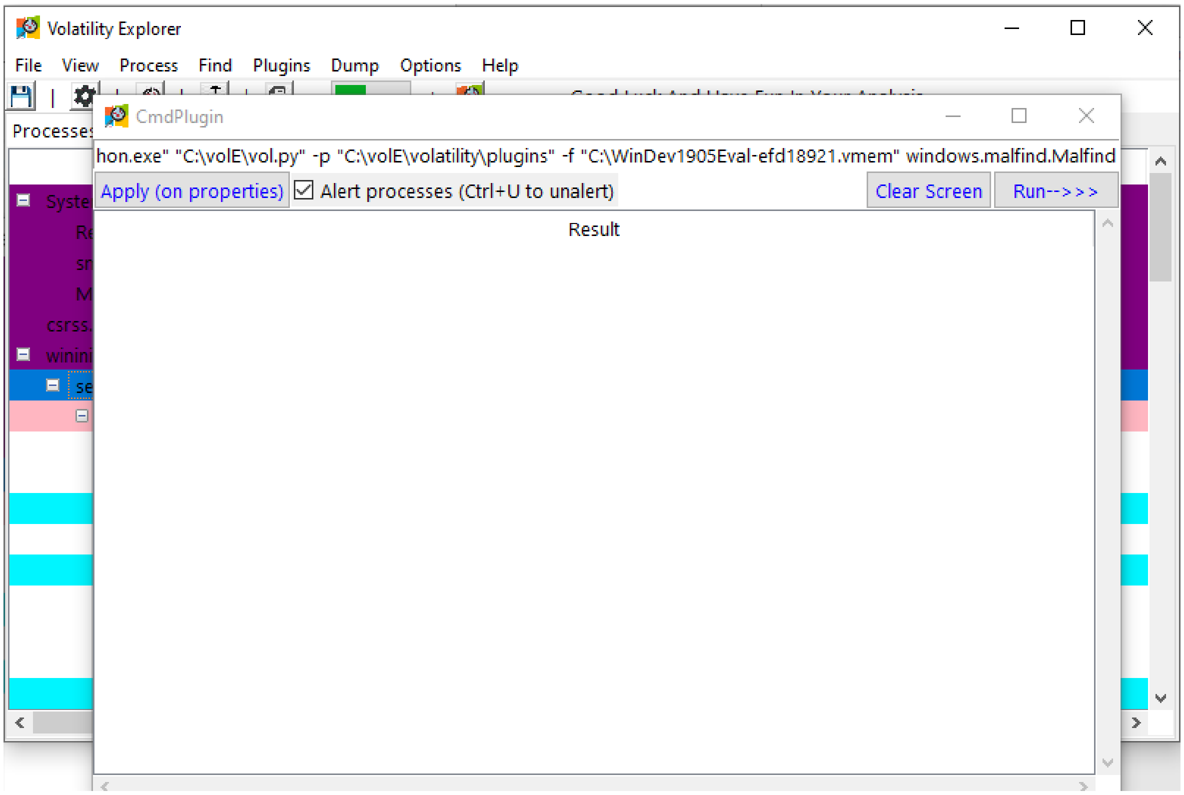
  For example running Struct Analyzer on _EPROCESS:



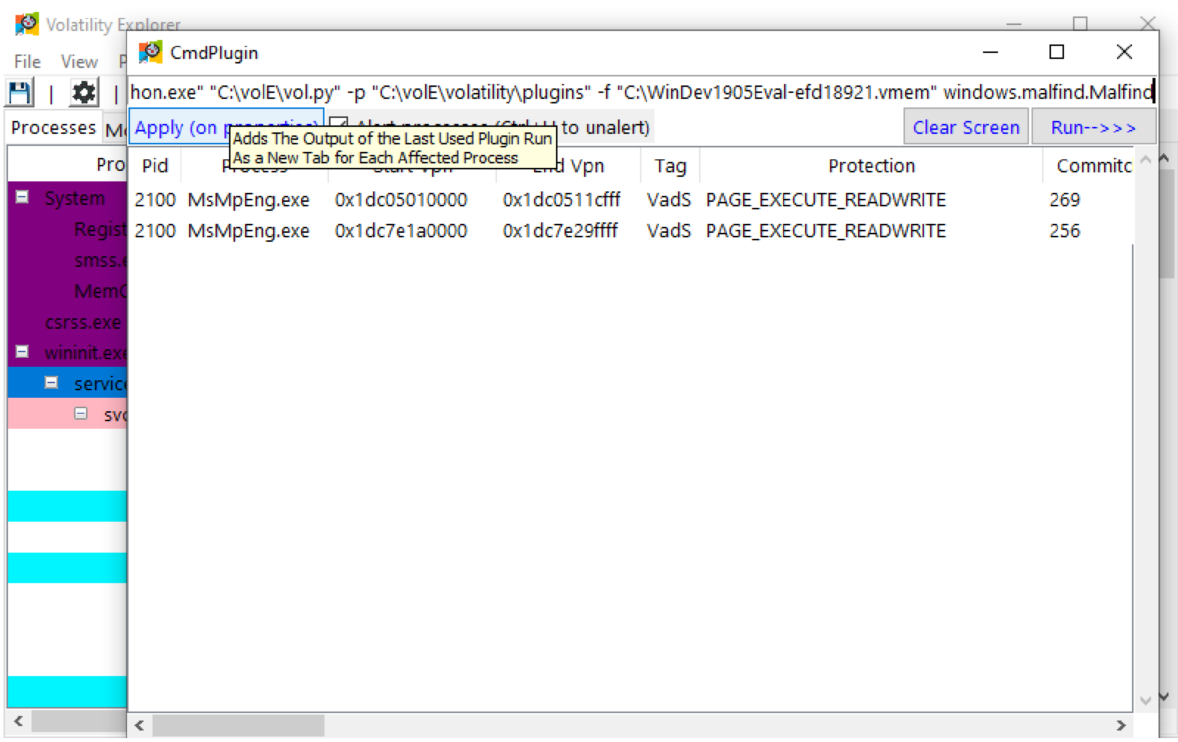
- Integrate With Other Plugins: This plugin has a lot of capabilities but analysts will probably have to use much more plugins during their work. This plugin allows you to easily integrate with other plugins by navigating to the Plugins tab->All/Well Known><plugin_name>:



It will automatically enter the command line (you can add arguments if you want to).

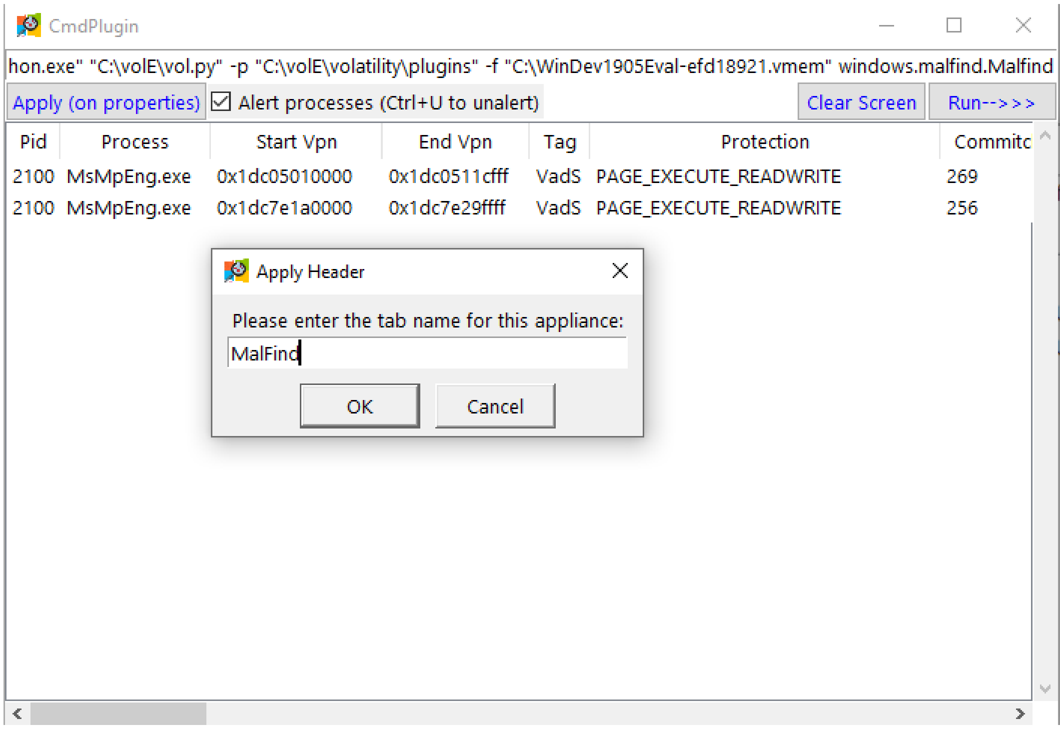


Clicking Run will run the plugin (as a subprocess)

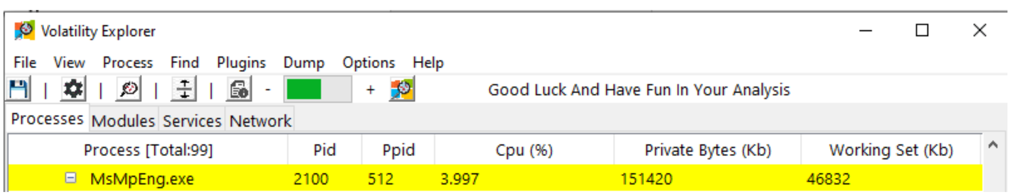


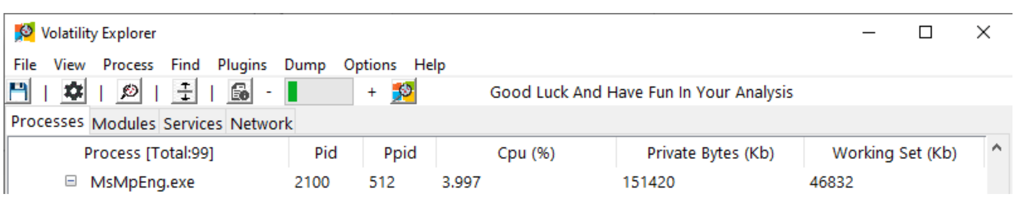
If we will click “Apply (on properties)” it will add the plugin output to the corresponding process properties and make a visual alert for this process.

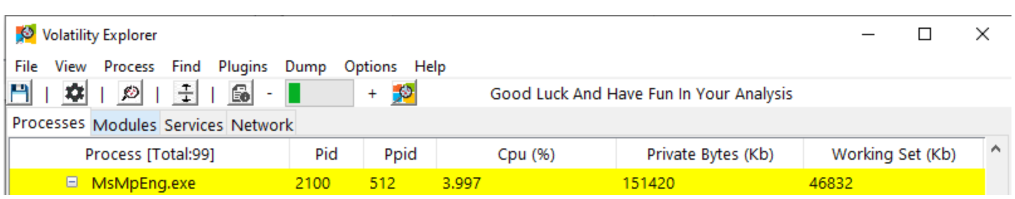
You will then have to enter a name for the new tab that will show in the properties window. For example, let's call it MalFind:



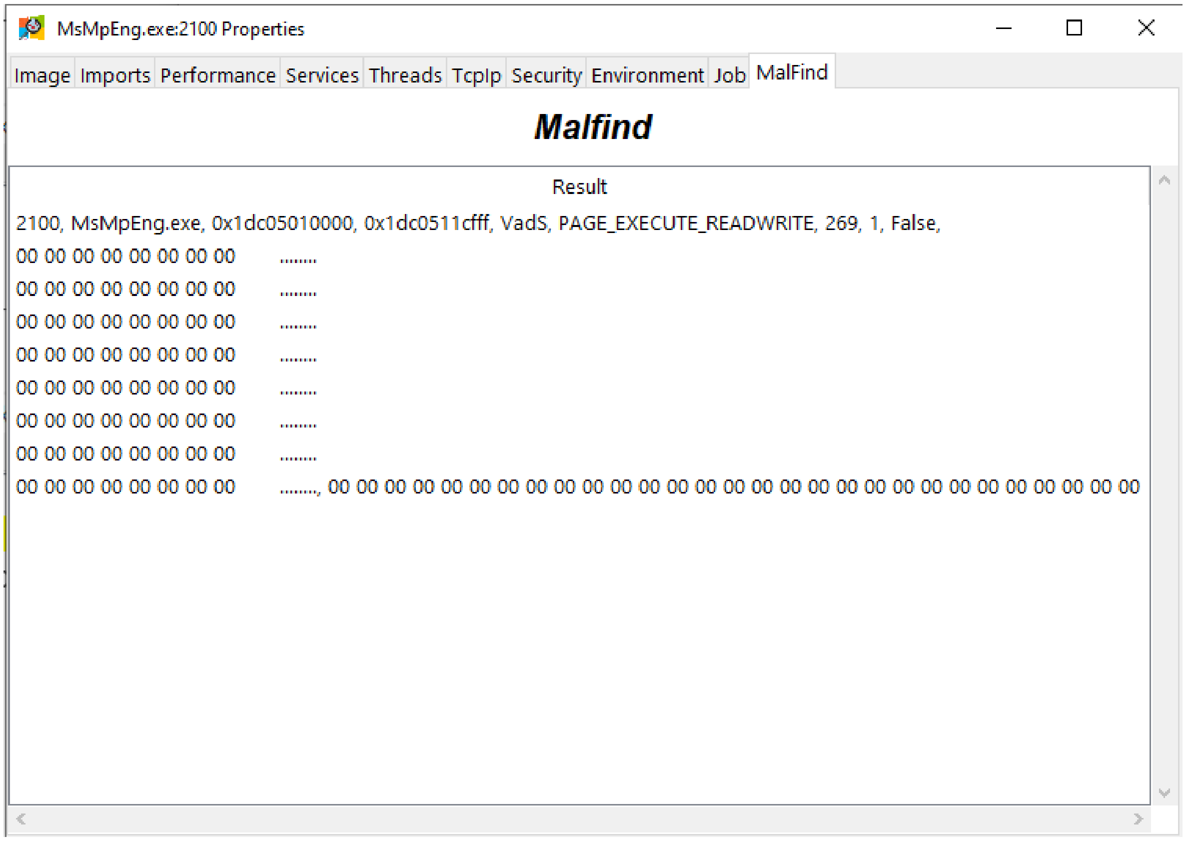
Which will make the process with PID 2100 flash as yellow:







When we double click this process (or Right Click->Properties) we will see a new tab - MalFind (the name we chose) tab with the results of the Malfind:



Once we enter the process the visual alert only for this process will stop (or you can unalert all the processes at once using the Menu->View->Unalert Processes (Ctrl+U)).

This is very strong capability which will let you organize your analysis using only the VolExp plugin, allowing you access to other plugins through it.

Plugin Flow:

This plugin starts by checking the user parameters, if something is missing it will pop up the options menu and ask the user to insert the missing data.

Then the plugin will get all processes data from the main table (PsTree view), and sort the processes in a PsTree view.

Next, it will run 2 additional processes (multiprocessing) to get data from the FileScanGui and WinObjGui plugins (MftParserGui will be added in the future when it will be added to the framework [the code for this already ready]).
At the same time its will start 5 additional threads, one for each of the following tasks:

1. Get Services using the SvcScan plugin and update the table.
2. Get Network Data using the NetScan plugin and update the table.
3. Get Process Handles using the Handles plugin (adding the Decoded Access as well) and update the table.
4. Get Process Environment Variables and Imports.
5. Attempt to translate the process SIDs to Username (using the GetSIDs plugin that I have written), and get additional information about each PE (Company Name, File Description, File Version, Internal Name, Legal Copyright, Original File Name, Product name, Product Version) using pefile module.
   After this last thread finishes, it will create X numbers of threads (1 thread for each hive), walk over its keys and get the registry data (to be used by RegEdit).

It is especially important to note the fact that the better your computer's hardware, the program will run faster. (If you try to run VolExp plugin using a very slow computer the UI may freeze sometimes, especially at the beginning, because a slow computer can't really handle 5 different threads and 2 additional processes so quickly.)

Struct Analyzer

Introduction

Struct Analyzer is a nice extension to the other UI plugins (but can also run as a plugin).
It parses structs remarkably similarly to volshell's dt function, but in an Explorer UI, so you can easily analyze structs in a graphical hierarchy.

Additionally, you can run all the volatility function on the struct you analyze and view the results. If the result is another struct, it will open in a new struct analyzer window. If it is a list/generator of structures, it will open a list of structs view with the ability to double click on one or more to open struct analyzer on the chosen struct.

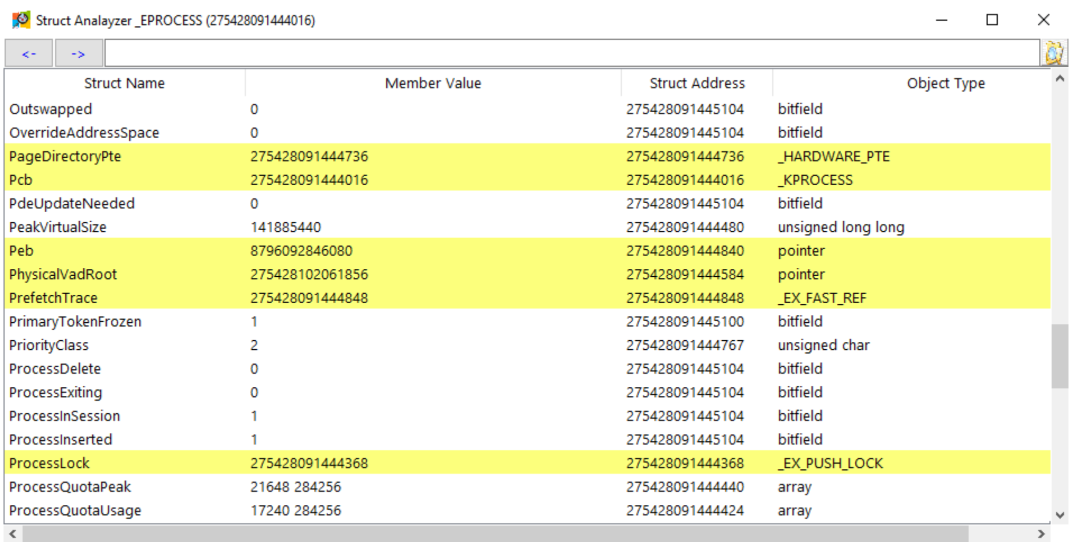
Motivation

Despite the fact that VolExp's utilities show us a lot of information, it cannot display everything - so I created Struct Analyzer, similarly to dt in WinDbg, it can parse structs inside the main struct by clicking at it), so I can view any additional information in any struct I wish, by displaying the struct in another view.

Arguments

This plugin accepts the following arguments:

- --STRUCT: the struct type to analyze.
- --ADDR: the address of the struct.
- --PID: the PID of the address space's process of the struct (enter "–PID=physical" for physical address space).



Struct Analayzer _EPROCESS (275428091444016)

Struct Name	Member Value	Struct Address	Object Type
Outswapped	0	275428091445104	bitfield
OverrideAddressSpace	0	275428091445104	bitfield
PageDirectoryPte	275428091444736	275428091444736	_HARDWARE_PTE
Pcb	275428091444016	275428091444016	_KPROCESS
PdeUpdateNeeded	0	275428091445104	bitfield
PeakVirtualSize	141885440	275428091444480	unsigned long long
Peb	8796092846080	275428091444840	pointer
PhysicalVadRoot	275428102061856	275428091444584	pointer
PrefetchTrace	275428091444848	275428091444848	_EX_FAST_REF
PrimaryTokenFrozen	1	275428091445100	bitfield
PriorityClass	2	275428091444767	unsigned char
ProcessDelete	0	275428091445104	bitfield
ProcessExiting	0	275428091445104	bitfield
ProcessInSession	1	275428091445104	bitfield
ProcessInserted	1	275428091445104	bitfield
ProcessLock	275428091444368	275428091444368	_EX_PUSH_LOCK
ProcessQuotaPeak	21648 284256	275428091444440	array
ProcessQuotaUsage	17240 284256	275428091444424	array

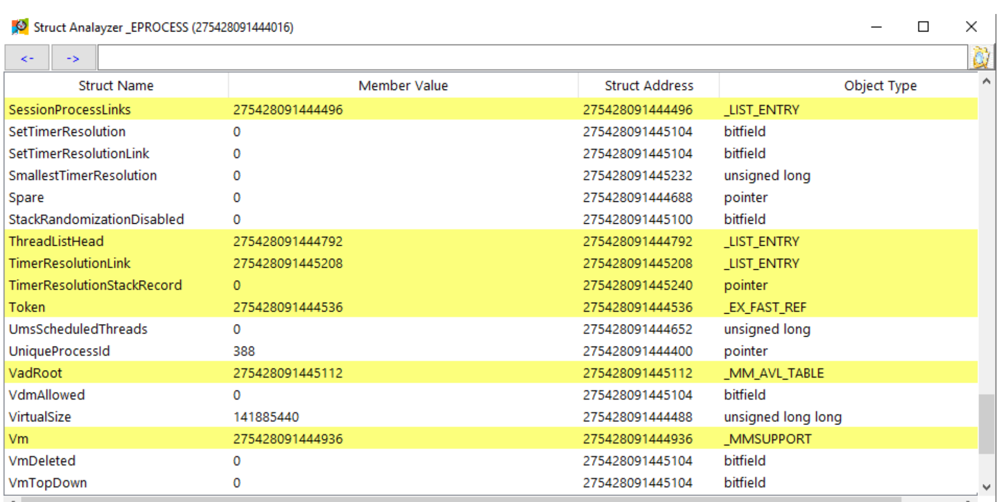
Some Research

The research of this plugin had a lot to do with how Volatility3's framework works and how to parse it correctly into a GUI where you can view all the data and run the functions you need – basically giving full support to all of Volatility's functions for the chosen struct, such as cast and dereference_as.

Abilities

Struct Analyzer is built on top of Explorer so everything you can do using Explorer, you can do with the Struct Analyzer window:

- Display all the struct members.

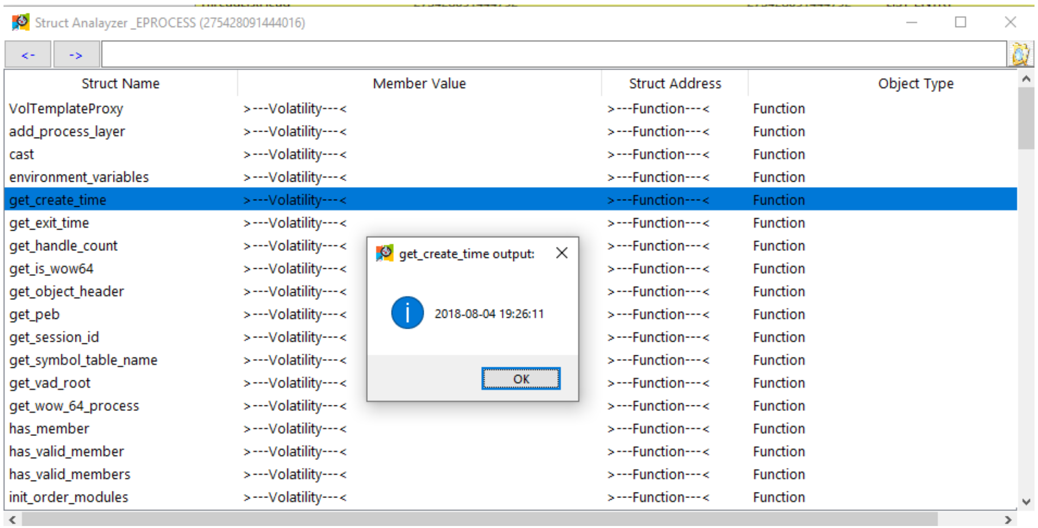


Struct Analayzer _EPROCESS (275428091444016)

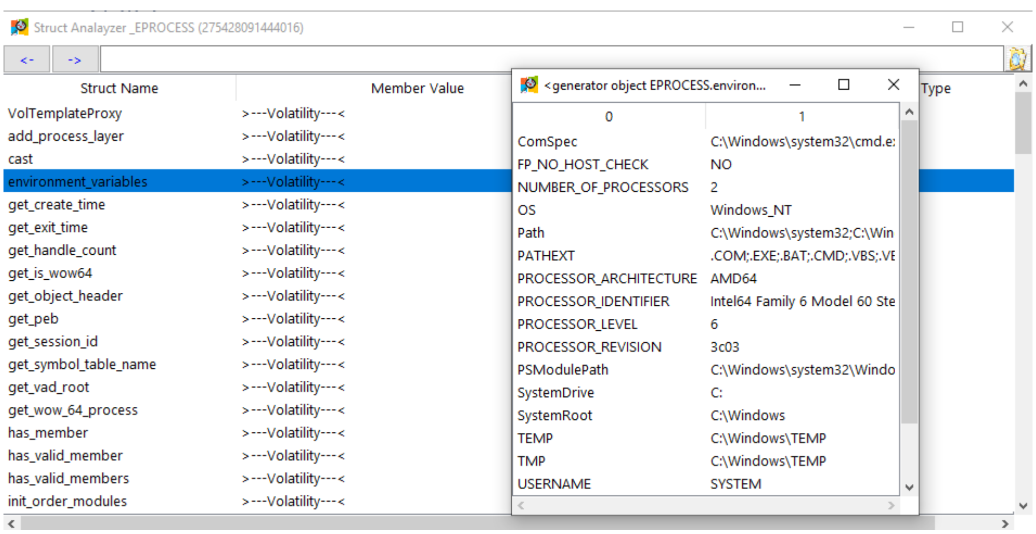
Struct Name	Member Value	Struct Address	Object Type
SessionProcessLinks	275428091444496	275428091444496	_LIST_ENTRY
SetTimerResolution	0	275428091445104	bitfield
SetTimerResolutionLink	0	275428091445104	bitfield
SmallestTimerResolution	0	275428091445232	unsigned long
Spare	0	275428091444688	pointer
StackRandomizationDisabled	0	275428091445100	bitfield
ThreadListHead	275428091444792	275428091444792	_LIST_ENTRY
TimerResolutionLink	275428091445208	275428091445208	_LIST_ENTRY
TimerResolutionStackRecord	0	275428091445240	pointer
Token	275428091444536	275428091444536	_EX_FAST_REF
UmsScheduledThreads	0	275428091444652	unsigned long
UniqueProcessId	388	275428091444400	pointer
VadRoot	275428091445112	275428091445112	_MM_AVL_TABLE
VdmAllowed	0	275428091445104	bitfield
VirtualSize	141885440	275428091444488	unsigned long long
Vm	275428091444936	275428091444936	_MMSUPPORT
VmDeleted	0	275428091445104	bitfield
VmTopDown	0	275428091445104	bitfield

- Display all the struct functions, with the ability to run them (and also insert arguments if necessary). The output will be displayed in a popup window of the following possible types (according to the result):

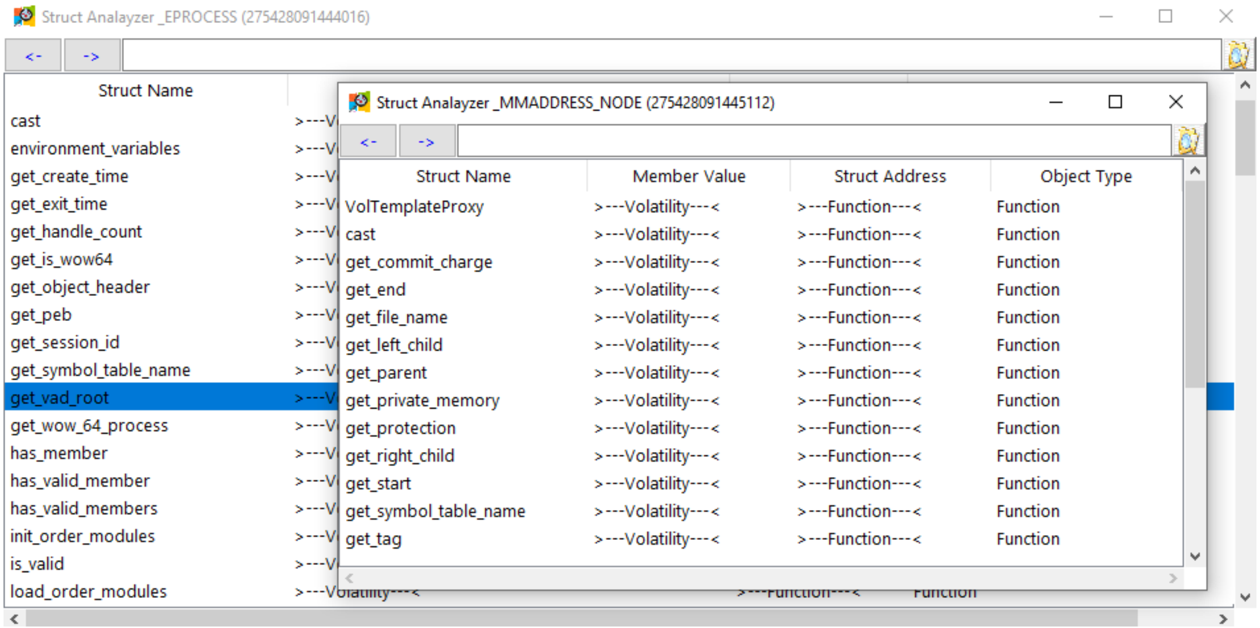
  1. As a string:



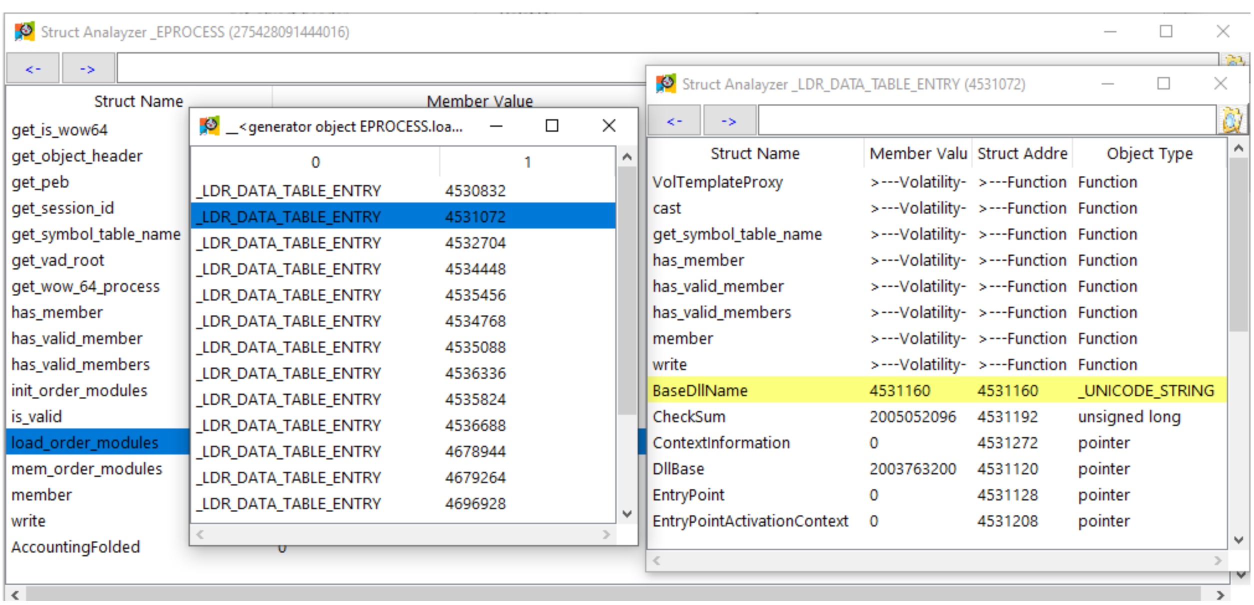
  2. A list of strings:



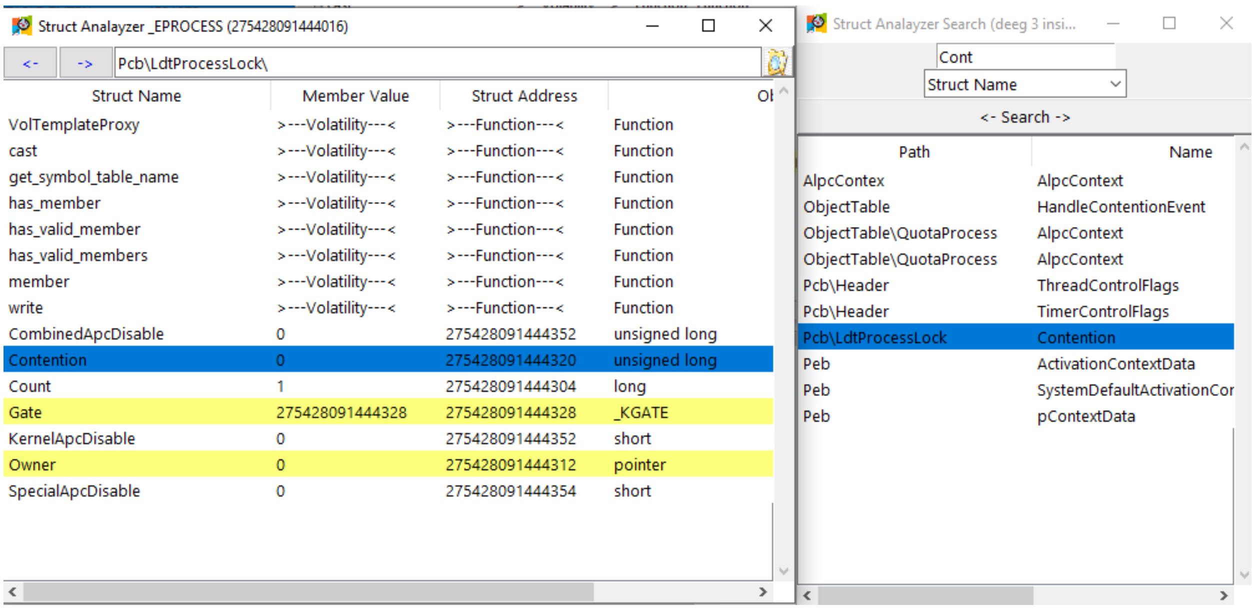
3. Object (display in another Struct Analyzer):



4. A list of objects (Double clicking on an element (struct) will open it in a new Struct Analyzer window):

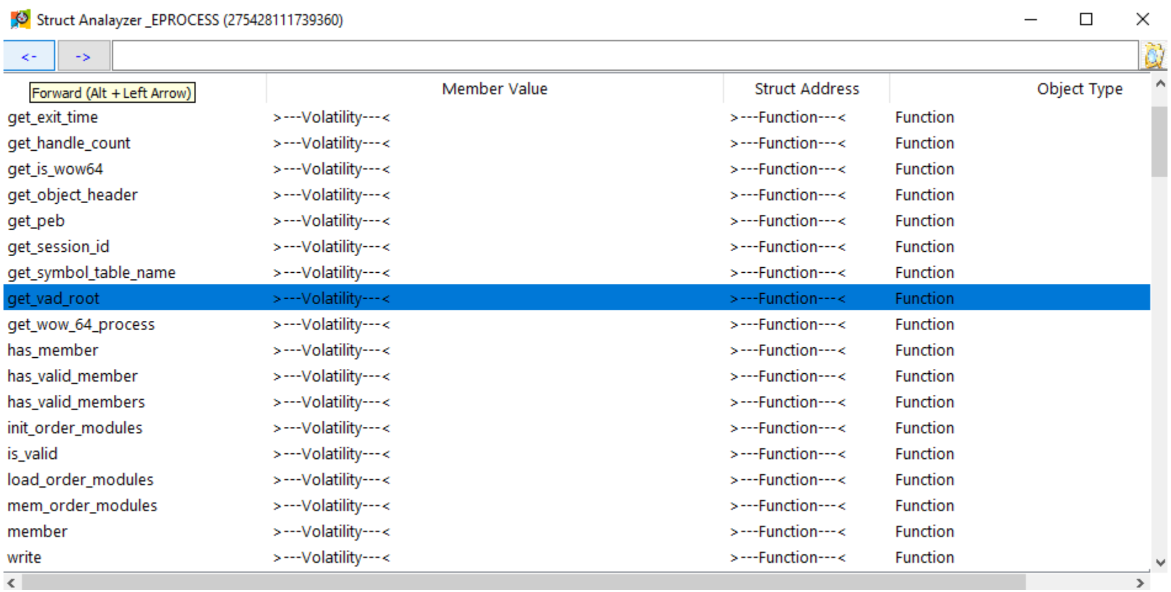


- By default, Struct Analyzer will parse up to 3 structs inside each struct (every time you get inside another struct it will parse 3 structs recursively inside as well). So you can use the explorer search to find elements 3 hierarchal levels lower.

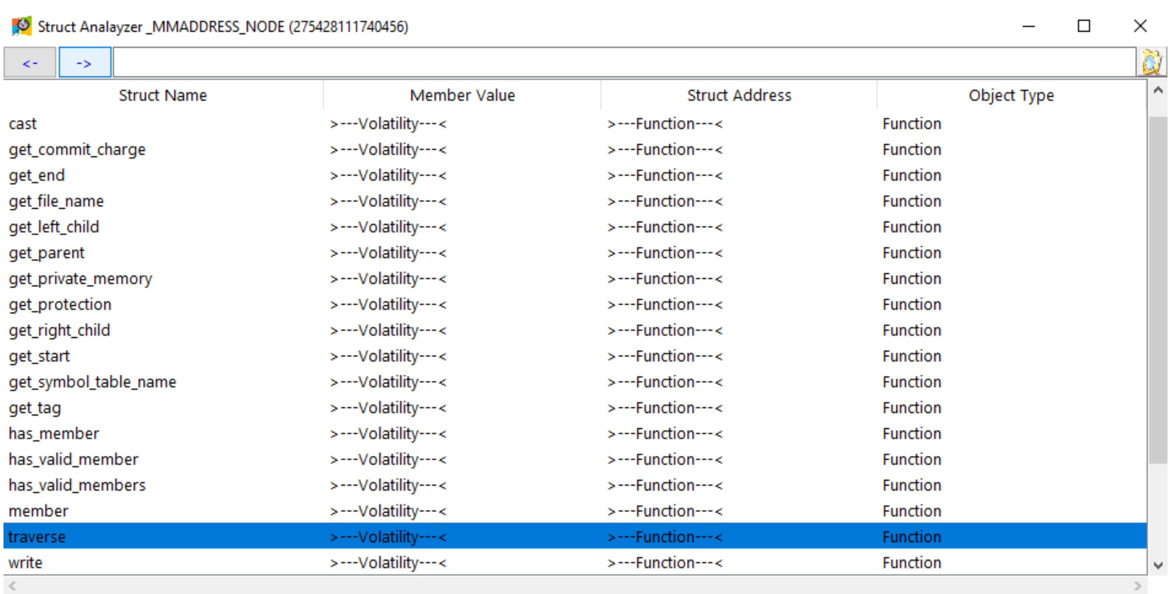


Struct Analyzer Flow (example)

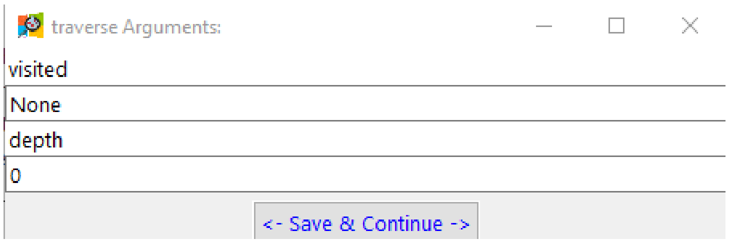
Here is an example of what you can do using Struct Analyzer (_PROCESS at address: 275428111739360):



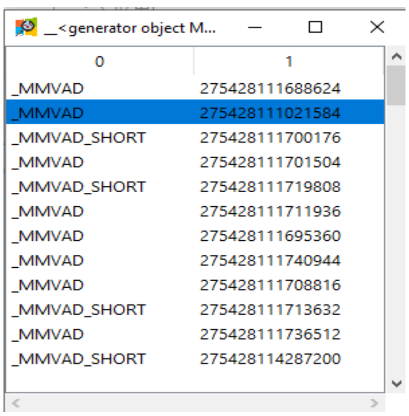
Double Clicking on the get_vad_root function will result in another struct analyzer window. This time, it will be a struct analyzer for _MMADDRESS_NODE (which is the struct type that is returned from the get_vad_root function):



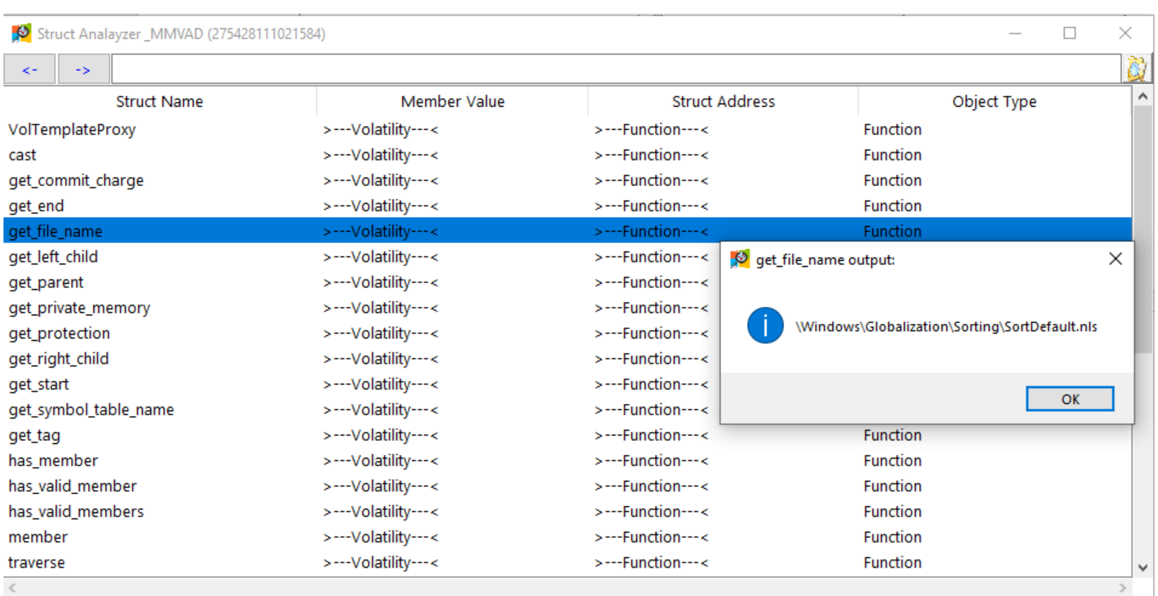
Double clicking “traverse” will result in the following popup, asking you to enter function parameters, since the traverse function requires parameters to run (this will happen every time you try to run a function that accepts arguments):



Double clicking traverse and entering the parameters (you probably shouldn't enter anything, just click Save & Continue button, since it fills in the default values) will result in the following list:

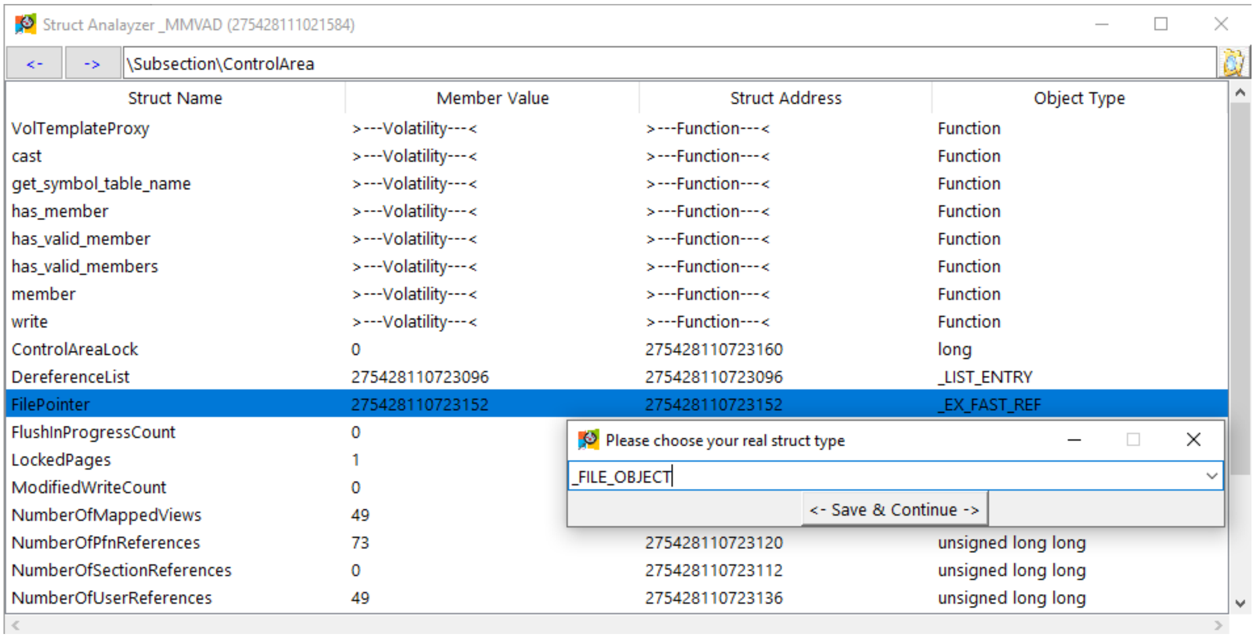


Double clicking on one of the structs will append another Struct Analyzer window of the chosen struct:

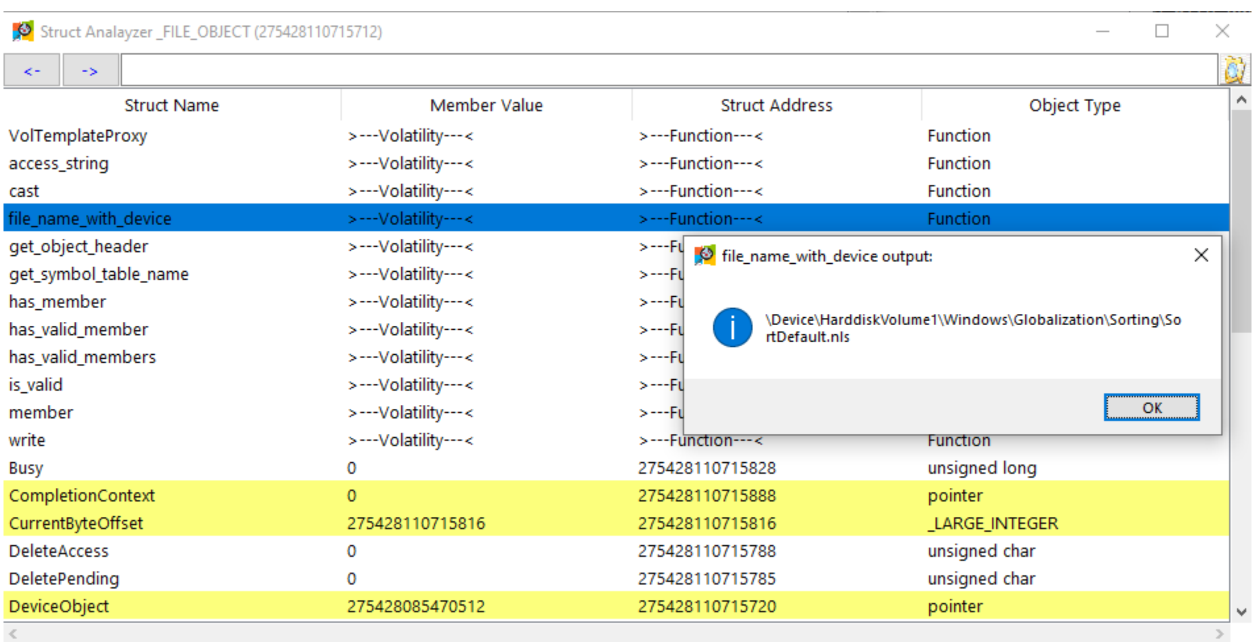


Double clicking on get_file_name will append the output of the function (the file name).

Let's go deeper and try to get the file name (without the get_file_name function). Navigating to Subsection->Control Area and then right clicking on the file and dereference it as _FILE_OBJECT:



This will result in another Struct Analyzer window for _FILE_OBJECT in the dereferenced address. Now let's click on the function file_name_with_device:



WinObjGui

Introduction

WinObjGui is a UI-based plugin which allows you to view all the kernel objects in an Explorer-like GUI.

It uses the WinObj plugin to parse the entire Windows object directory.

In addition to all the Explorer functions, you will also be able to integrate with Struct Analyzer and analyze structs from the object directory.

Motivation

After creating VolExp, which is remarkably similar to Process Explorer if we compare it to the SysInternals tool, I was looking for other UI tools that could help an analyst. Another SysInternals tool caught my eye: WinObj, so I decided to create this tool.

Furthermore, I realized that it can fit very well with the Explorer UI, and help make sure that the analyst doesn't miss anything (the output of the winobj.py has a lot of verbose data, making it easy to miss critical information using the CLI tool, so a GUI-based tool can help tremendously).

Startup Arguments

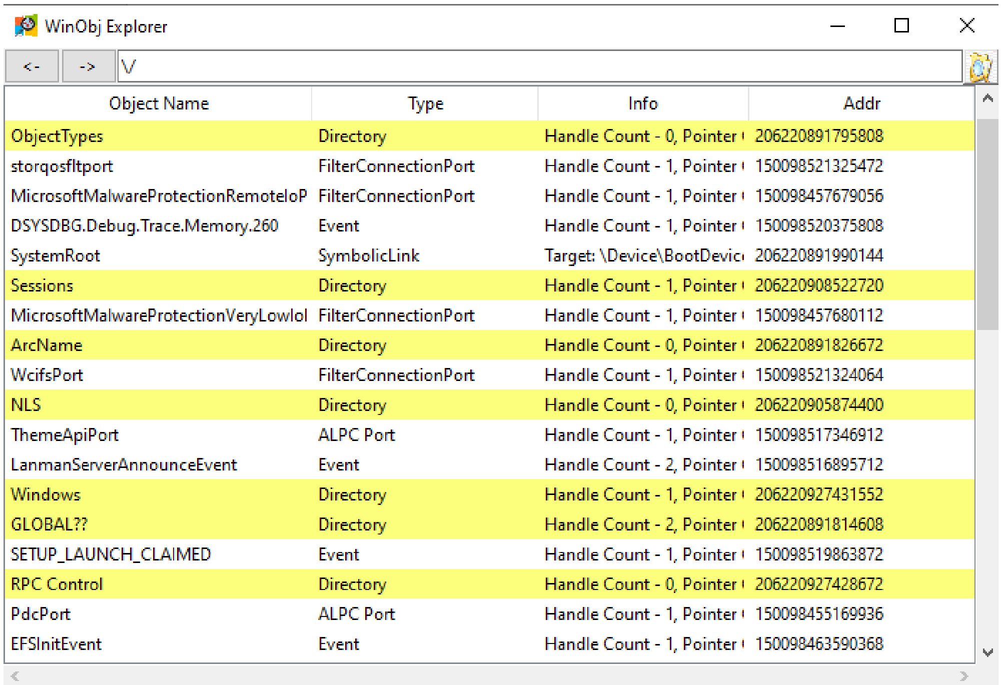
This plugin accepts the following argument:

- --GET-DICT: File path to recursively output all the windows directory data in a pickle format (this argument is optional, and is used by volexp. If this argument is present, then no GUI will appear).

Some Research

This Plugin had nearly no research, as it's just a UI view of the WinObj plugin.

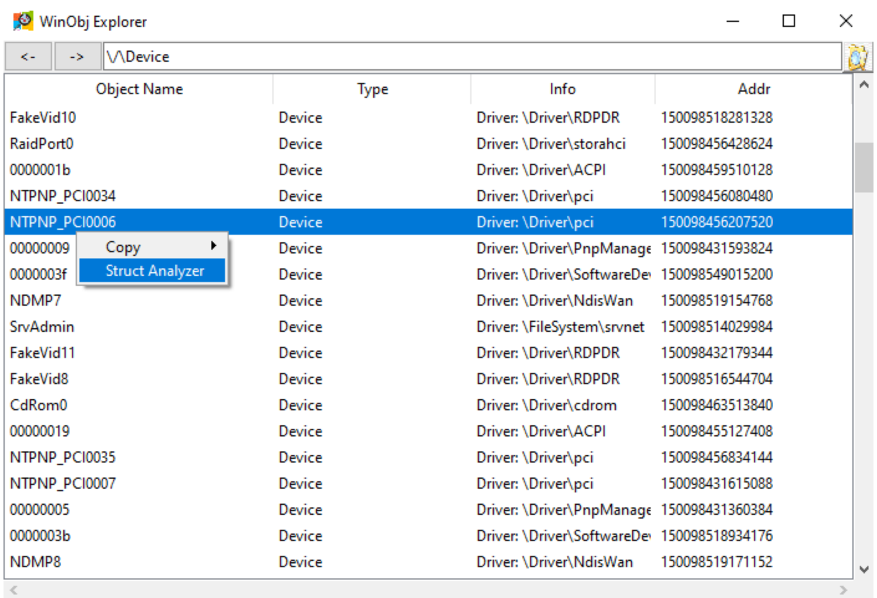
The only research that was necessary was to find out the object type of objects referenced by pointers. For example: driver -> _DRIVER_OBJECT, event -> _KEVENT, mutant -> _KMUTANT, etc.

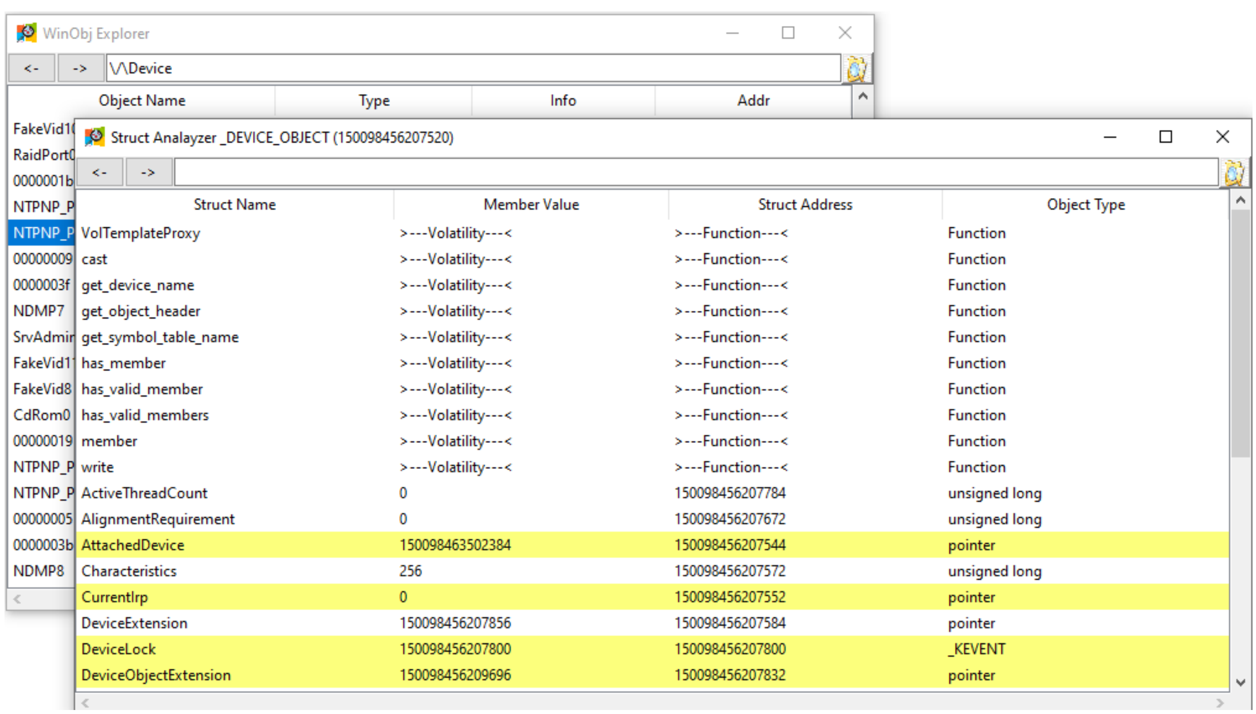


Abilities

WinObjGui is built on top of Explorer so everything you can do using [Explorer](), you can do with the WinObjGui Window:

- *Integrate with Struct Analyzer* – WinObjGui can integrate with [Struct Analyzer]() using right click on some struct and then click Struct Analyzer will result a new Struct Analyzer Window of the corresponding struct





FileScanGui

Introduction

FileScanGui is a UI plugin which allows you to view all the files in a Windows Explorer-Like UI. It also shows data pulled from the user assist (so you can know what files a certain user runs).

In addition to all the Explorer function you will also be able to integrate with Struct Analyzer and analyze _FILE_OBJECT structs and also dump files/directories or view a file's HexDump.

Motivation

While working on the VolExp plugin, I realized that VolExp is not enough.
During a regular system analysis, when using Process Explorer\Hacker you are also using another UI that Microsoft has created, which also seems to get the least credit – Windows Explorer, since it increases comfortability and ease of use tremendously. So I decide to create one.
It could help the user to navigate directly to the directory/file they want to and view/dump it easily.

Startup Arguments

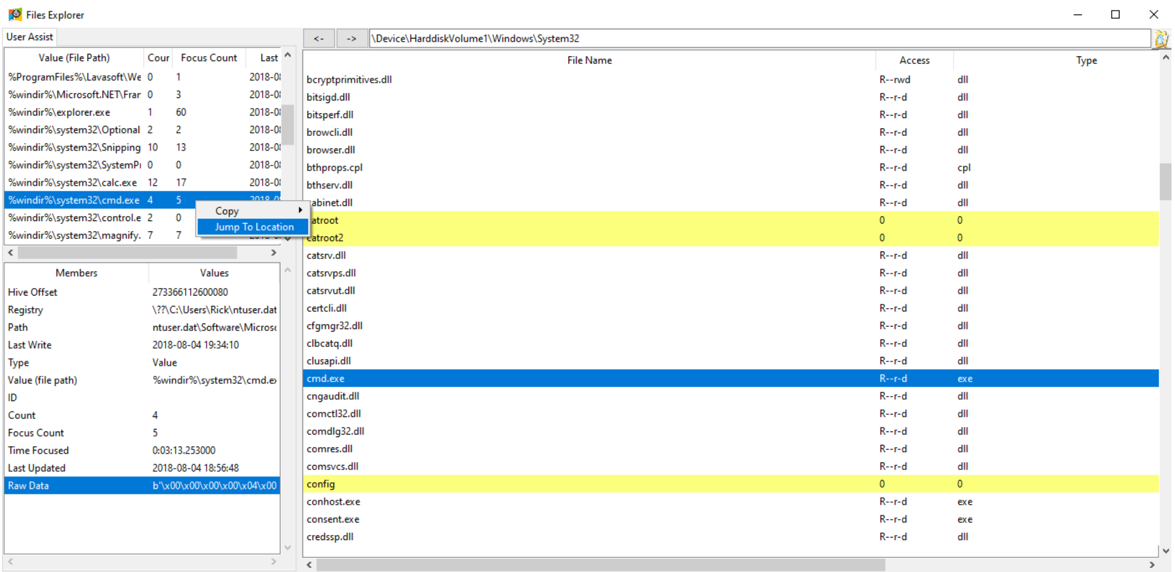
This plugin accepts the following arguments:

- --DUMP-DIR: Directory to dump files.
- --GET-DICT: File path to output all the filescan data, ordered by directories, in a pickle format (this argument is optional, and is used by volexp. If this argument is present, then no GUI will appear).

Some Research

My research for this plugin was realizing how to dump a file from memory to disk.
This is not something new and it was present in Volatility2 as well as in Rekall - but not in

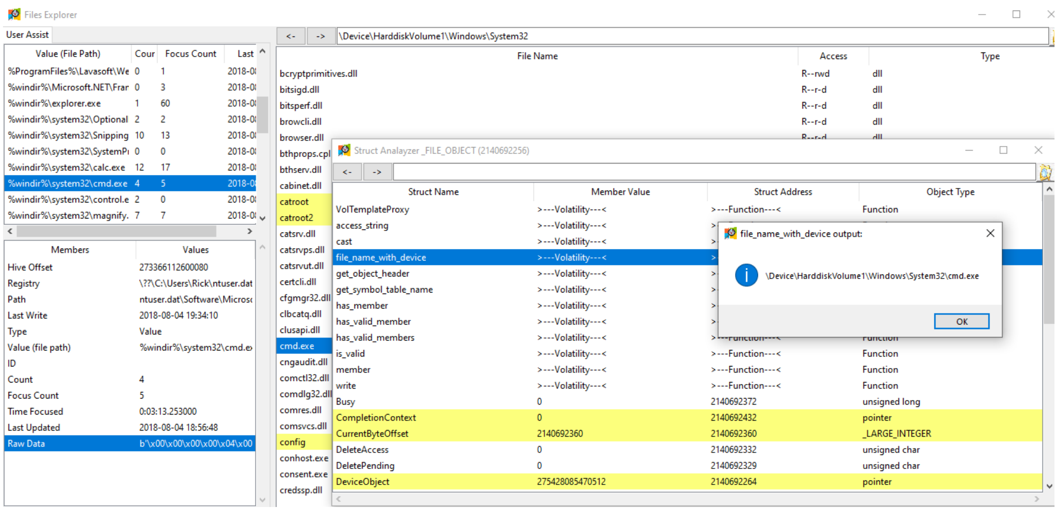
Volatility3, so I had to recreate it in my plugin. So I create this functionality using Volatility2's dumpfiles and some additional resources (Supports dump from both _control_area and _shared_cache_map).



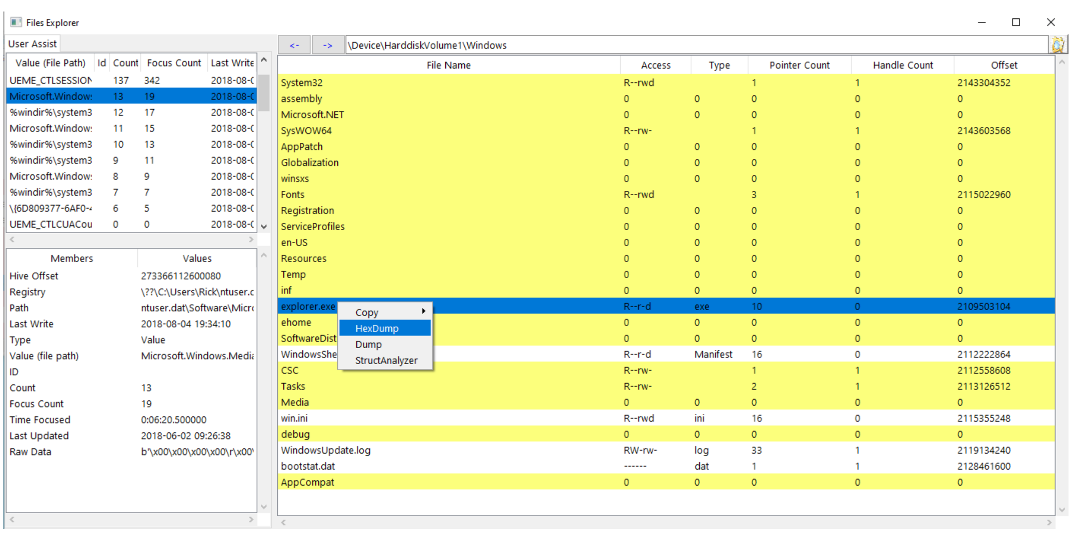
Abilities

FileScanGui is built on top of Explorer so everything you can do using [Explorer](), you can do with the FileScanGui Window:

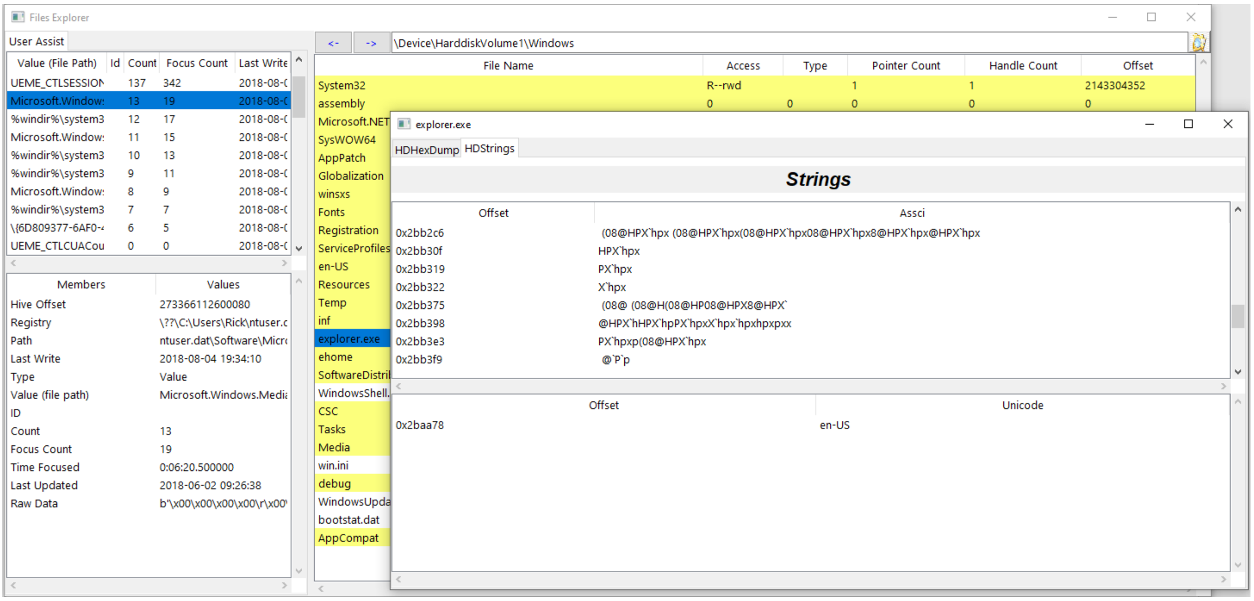
- *Integrate with Struct Analyzer* – FileScanGui can integrate with Struct Analyzer using right click on a file and then clicking Struct Analyzer will result a new Struct Analyzer Window of the corresponding _FILE_OBJECT
  For example, running Struct Analyzer on cmd.exe and running the file_name_with_device function inside the Struct Analyzer window:



- *Dump File/Directory* – FileScanGui also has a feature to dump files or directories using right click and “Dump” or “HexDump”:



Let’s see HexDump Strings view for example:



Pfn.py

Motivation

Most of memory forensics tools will create a view of something from the virtual address space of a process. Most, like Volatility, Rekall, etc. have a built-in function to translate virtual address to physical address, like the processor always does. But except for rammap from SysInternals there is not a working tool (that works on newer versions of Windows as well) that can translate physical address to virtual address or can give us forensics artifacts based on a given physical address.
Rekall does have this ability, but it fully works only for Windows version 8.0 and prior, and only on 64bit systems.

I realize that Windows still needs the ability to translate physical address to virtual address so I decide to take the challenge, do a lot of research and create this plugin (fully supported from Windows XP to Windows 10!), I decided to make it a suite of plugins: P2V, PFNInfo (that uses P2V) and rammap, that uses them both.

Rammap will give information for each physical page in the memory dump, so make sure to use the 64bit version of python if you run rammap on a large memory dump.

P2V – Physical to Virtual

Intruduction

This plugin translates a physical address to a virtual one using the pfndb.

It also tries to find the owner of the page, as well as if there is any file mapped to this address - and if there is, which processes mapped it.

Startup Arguments:

This plugin accepts only one argument:

- -- ADDRESS: the physical address to translate.

```
c:\volE>vol.py -f C:\Users\[redacted]\Downloads\OtterCTF\OtterCTF.vmem windows.pfn.P2V -
-ADDRESS 0x20000
Volatility 3 Framework 1.2.1-beta.1
Progress:   41.02               Scanning primary2 using PdbSignatureScanner
Owner   Loading To      File Name       File Offset     Physical        Virtual

WebCompanionIn (3880), WebCompanion.e (3856)    3880, 3856      \Device\HarddiskVol
ume1\Windows\assembly\NativeImages_v2.0.50727_32\mscorlib\62a0b3e4b40ec0e8c5cfaa0c8
848e64a\mscorlib.ni.dll       0x399400        0x20000 0x7336b000, 0x7336b000

c:\volE>
```

Some research

To understand how this plugin works we need to understand the _MMPFN struct.

To help you better understand, let's define PFN as "VAD for the physical layer" – it's a struct that has a lot of data about the physical page.

```
(primary2) >>> dt("_MMPFN")
nt_symbols1!_MMPFN (48 bytes)
  0x0 :   u1                        nt_symbols1!__unnamed_152f
  0x8 :   u2                        nt_symbols1!__unnamed_1531
 0x10 :   Lock                      nt_symbols1!long
 0x10 :   PteAddress                nt_symbols1!pointer
 0x10 :   PteLong                   nt_symbols1!unsigned long long
 0x10 :   VolatilePteAddress        nt_symbols1!pointer
 0x18 :   u3                        nt_symbols1!__unnamed_1536
 0x1c :   UsedPageTableEntries      nt_symbols1!unsigned short
 0x1e :   VaType                    nt_symbols1!unsigned char
 0x1f :   ViewCount                 nt_symbols1!unsigned char
 0x20 :   AweReferenceCount         nt_symbols1!long
 0x20 :   OriginalPte               nt_symbols1!_MMPTE
 0x28 :   u4                        nt_symbols1!__unnamed_153e
```

This struct doesn't come alone. The OS has an array of _MMPFNs that we can find in MmPfnDatabase (a pointer to the start of the array).

The first entry of the array describes the first physical page, the second describe the second and so on… In short, the array length = ram size / page_size. In 64 bit systems this struct 0x30 bytes large, meaning that for every page the size of 0x1000, 0x30 of memory will be immediately taken.

Thus, this struct takes more 1% from your memory, and never pages out, but there are some nonvalid _MMPFNs and we can check whether a PFN is valid or not using the MiPfnBitMap (on some versions).

```
addr = nt.get_symbol('MiPfnBitMap').address
rtl = nt.object("_RTL_BITMAP", addr)

# Check if the page index not out of range
if rtl.SizeOfBitMap < pfn_index:
    return False

# Check if the page is valid
c_byte = context.layers['primary'].read(rtl.Buffer + (pfn_index >> 3), 1)[0]
if c_byte & (2 ** (pfn_index % 8)) == (2 ** (pfn_index % 8)):
    return True
return False
```

Or check the _MMPFN.u4.PfnExists flag (on Windows versions that the bitmap is not present).

So how do we make use of this struct?

The OS uses this struct to track the physical pages – whether the page is in use, the address of the page, the virtual(_MMPFN.PteAddress) and physical address(_MMPFN.u4.PteFrame << PAGE_BITS | (_MMPFN.PteAddress & 0xFFF)) of the PTE (that manages this page), the OriginalPte (to handle soft page faults) and much more.

So if each _MMPFN contains information about the PTE that manages the page, we can just go to the PTE that manages our _MMPFN, the PTE that manages the PTE that manages the _MMPFN and so on… Basically, we have to reverse the entire steps of the virtual address translation.

After we go through all that, we end in the DirectoryTableBase of the process, so I created a dictionary that maps DTB address to the process and follows this routine, which allows me to get both the virtual address and the process that mapped this page.

To make sure that this works, let's go to volshell and check the values from the example on the start:

```
c:\volE>vol.py -f C:\Users\[redacted]\Downloads\OtterCTF\OtterCTF.vmem windows.pfn.P2V -
-ADDRESS 0x20000
Volatility 3 Framework 1.2.1-beta.1
Progress:   41.02               Scanning primary2 using PdbSignatureScanner
Owner   Loading To      File Name       File Offset     Physical        Virtual

WebCompanionIn (3880), WebCompanion.e (3856)    3880, 3856      \Device\HarddiskVol
ume1\Windows\assembly\NativeImages_v2.0.50727_32\mscorlib\62a0b3e4b40ec0e8c5cfaa0c8
848e64a\mscorlib.ni.dll       0x399400        0x20000 0x7336b000, 0x7336b000

c:\volE>
```

```
(primary2) >>> pr.UniqueProcessId
3880
(primary2) >>> layer = pr.add_process_layer()
(primary2) >>> context.layers[layer].translate(0x7336b000)
(131072, 'memory_layer')
(primary2) >>> hex(131072)
'0x20000'
(primary2) >>>
```

And we got what we expected, the address 0x7336b000 translates to the physical address 0x20000 that we checked before (in the example on top).

But I actually lied to you guys, the example above didn't work in the algorithm explained above, because that method only works for pages that map to a specific process; it would not work for shared pages - if we try the method above we end up with the System (4) process, not with PID3880 and PID3856. So how did I get this process?

When more than one process wants to use the same page, the OS creates a "prototype" PTE (for files for example). We can check whether a PFN points to a prototype PTE using the _MMPFN.u3.PrototypePte flag. This type of PTE points to a _SUBSECTION struct.
By iterating through all the processes and collecting all the subsections that loaded into each process, we can then determine which process belongs to each subsection we find.

```python
def map_process_vads_subsections(self):
    """
    get all vads and subsections from a processes
    """
    self.subsections = {}
    self.vads = {}
    # Go all over the processes and collect all the subsections.
    for c_proc in pslist.PsList.list_processes(context = self.context,
                                               layer_name = self.config['primary'],
                                               symbol_table = self.config['nt_symbols']):

        # Go all over vads.
        for vad in c_proc.get_vad_root().traverse():
            if hasattr(vad, "Subsection") and vad.Subsection:
                try:
                    subsection = vad.Subsection
                    seen_subs = []
                    # Walk the subsection list
                    while subsection not in seen_subs and subsection:
                        seen_subs.append(subsection)
                        start_addr = subsection.SubsectionBase.real

                        # Vads sould not be in userspace
                        if start_addr > self.HighestUserAddress:

                            end_addr = start_addr + (subsection.PtesInSubsection * self.ntkrnlmp.get_type('_MMPTE').size)
                            range = (start_addr, end_addr)
                            if not range in self.subsections:
                                self.subsections[range] = []
                            self.subsections[range].append((c_proc, vad, subsection))
                        subsection = subsection.NextSubsection
                except exceptions.InvalidAddressException:
                    pass
                if not int(c_proc.UniqueProcessId) in self.vads:
                    self.vads[int(c_proc.UniqueProcessId)] = []
                self.vads[int(c_proc.UniqueProcessId)].append((vad.get_start(), vad.get_end(),vad))
```

(The code above shows how we map all the subsection ranges to the self.subsection dictionary.)

Abilities

- Translate physical address to virtual address
- Find the processes that map a given physical address
- Find a file mapped in a given physical address (if present)

PFNInfo – page frame number information

Introduction

As its name suggests, its main goal is to bring as much information as possible from a given physical address using the PFN Database.

Motivation

After creating P2V, RAMMAp from SysInternals was yet stronger than my own tool. I needed to find much more information, like the use of a page, page priority, page list, etc...

For that reason, I decided to create a new plugin with this goal in mind: To give additional information about the page, not just the owner and the virtual address like P2V already does, but additional possible information we can get.

Startup Arguments

This plugin accepts one the following arguments:

- --ADDRESS: Address of the _MMPFN struct
- --INDEX: Index of the _MMPFN struct (inside the array [the page frame number])
- --PAGE: Address of a page

Note: --INDEX and –PAGE are relatively similar. If you use –PAGE={address} you can use –INDEX={address}/PAGE_SIZE.

Some research

In the previous plugin (P2V), a part of the research was to investigate the _MMPFN struct and how the OS uses it. Here, I had to do much more research, in different Windows versions, such as how to extract the page list, priority, reference and share count, page color(numa), PTE type, NxBit, etc.

I had to find how to locate general memory areas, like NonPagePool and PagePool (for session space as well) in order to identify the use of a page.

```
c:\volE>vol.py -f C:\Users\[redacted]\Downloads\OtterCTF\OtterCTF.vmem windows.pfn.pfninfo --INDEX 100000
Volatility 3 Framework 1.2.1-beta.1
Progress:   41.02               Scanning primary2 using PdbSignatureScanner
PFN Index       PFN Address     Page List       Priority        Reference       Share Count     Page Color      Pte Type        NxBit   Use     File Name       Offset  Image   Pool Tags

100000  18446738026400398848    Active  5       1       1       0       Prototype PTE   1       Mapped File     \Device\HarddiskVolume1\Windows\System32\winevt\Logs\Microsoft-Windows-Application-Experience%4Program-Compatibi
lity-Troubleshooter.evtx        65536   False   ([],)

c:\volE>
```

Abilities

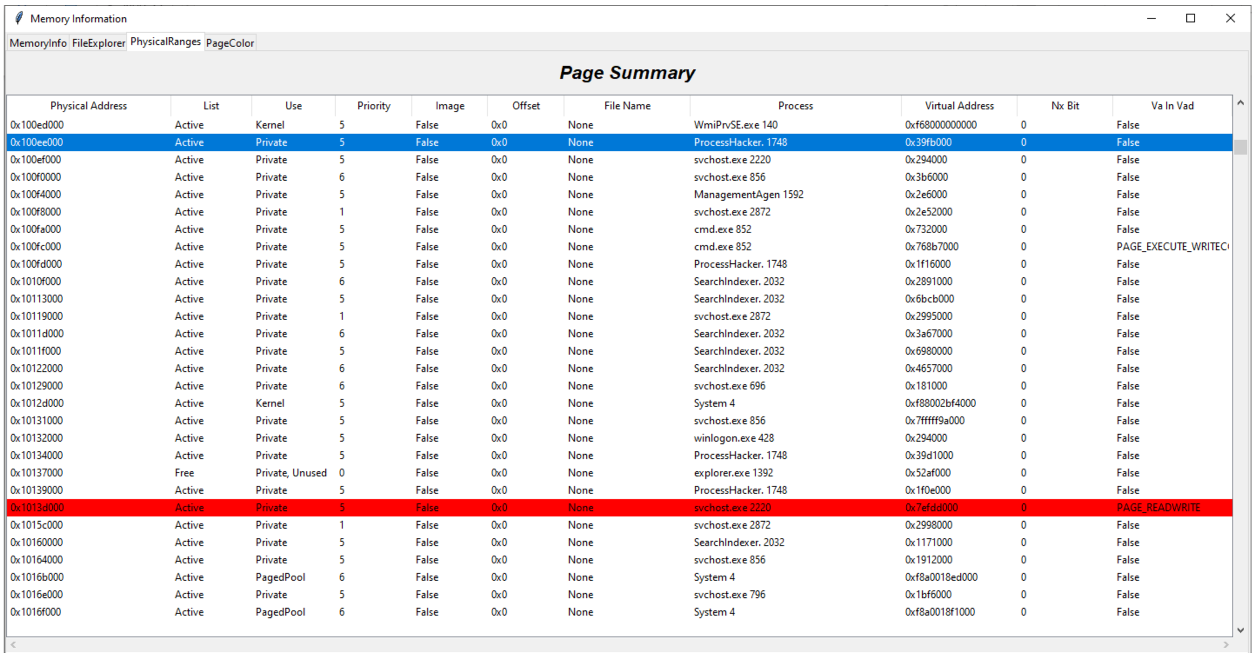
This plugin lists a lot of information related to a physical page, including:

PFN index, PFN address, page list, priority, reference and share count, page color(numa), PTE type, NxBit, the use of a page, file name (if any), file offset, is the file an image and pool tags (scan the physical address from built in pool tags list)

RAMMap – Random Access Memory Mapping

Introduction

This plugin is the reason that P2V and PFNInfo exists. RAMMap uses them both to get every information we can get from a given physical page, in addition to attempting to mark pages it finds suspicious.



Physical Address	List	Use	Priority	Image	Offset	File Name	Process	Virtual Address	Nx Bit	Va In Vad
0x100ed000	Active	Kernel	5	False	0x0	None	WmiPrvSE.exe 140	0xf68000000000	0	False
0x100ee000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x39fb000	0	False
0x100ef000	Active	Private	5	False	0x0	None	svchost.exe 2220	0x294000	0	False
0x100f0000	Active	Private	6	False	0x0	None	svchost.exe 856	0x3b6000	0	False
0x100f4000	Active	Private	5	False	0x0	None	ManagementAgen 1592	0x2e6000	0	False
0x100f8000	Active	Private	1	False	0x0	None	svchost.exe 2872	0x2e52000	0	False
0x100fa000	Active	Private	5	False	0x0	None	cmd.exe 852	0x732000	0	False
0x100fc000	Active	Private	5	False	0x0	None	cmd.exe 852	0x768b7000	0	PAGE_EXECUTE_WRITEC
0x100fd000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x1f16000	0	False
0x1010f000	Active	Private	6	False	0x0	None	SearchIndexer. 2032	0x2891000	0	False
0x10113000	Active	Private	5	False	0x0	None	SearchIndexer. 2032	0x6bcb000	0	False
0x10119000	Active	Private	1	False	0x0	None	svchost.exe 2872	0x2995000	0	False
0x1011d000	Active	Private	6	False	0x0	None	SearchIndexer. 2032	0x3a67000	0	False
0x1011f000	Active	Private	5	False	0x0	None	SearchIndexer. 2032	0x6980000	0	False
0x10122000	Active	Private	6	False	0x0	None	SearchIndexer. 2032	0x4657000	0	False
0x10129000	Active	Private	6	False	0x0	None	svchost.exe 696	0x181000	0	False
0x1012d000	Active	Kernel	5	False	0x0	None	System 4	0xf88002bf4000	0	False
0x10131000	Active	Private	5	False	0x0	None	svchost.exe 856	0x7fffff9a000	0	False
0x10132000	Active	Private	5	False	0x0	None	winlogon.exe 428	0x294000	0	False
0x10134000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x39d1000	0	False
0x10137000	Free	Private, Unused	0	False	0x0	None	explorer.exe 1392	0x52af000	0	False
0x10139000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x1f0e000	0	False
0x1013d000	Active	Private	5	False	0x0	None	svchost.exe 2220	0x7efdd000	0	PAGE_READWRITE
0x1015c000	Active	Private	1	False	0x0	None	svchost.exe 2872	0x2998000	0	False
0x10160000	Active	Private	5	False	0x0	None	SearchIndexer. 2032	0x1171000	0	False
0x10164000	Active	Private	5	False	0x0	None	svchost.exe 856	0x1912000	0	False
0x1016b000	Active	PagedPool	6	False	0x0	None	System 4	0xf8a0018ed000	0	False
0x1016e000	Active	Private	5	False	0x0	None	svchost.exe 796	0x1bf6000	0	False
0x1016f000	Active	PagedPool	6	False	0x0	None	System 4	0xf8a0018f1000	0	False

(Colored in red because the VAD is marked as PAGE_READWRITE but the NxBit is unset).

Startup Arguments

This plugin accepts the following arguments:

- --ADDRESS: Starting address (defaults to 0)
- --SIZE: The size to analyze (the default is the memory dump size minus the starting address)
- --COLORED: Display only colored pages (defaults to false)

Some Research

I researched and looked for "weird" page states, categorizing them and giving each of them a different color.

For example, it will catch a conflict between the VAD protection to the PTE protection.

```
\NDLE WINAPI LoadRemoteLibraryR( HANDLE hProcess, LPVOID lpBuffer, DWORD dwLeng

  lpRemoteLibraryBuffer = VirtualAllocEx( hProcess, NULL, dwLength, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE );
  lpRemoteLibraryBuffer = VirtualAllocEx(hProcess, NULL, dwLength, MEM_RESERVE | MEM_COMMIT, PAGE_READONLY);
  DWORD oldPrection;
  VirtualProtectEx(hProcess, lpRemoteLibraryBuffer, dwLength, PAGE_EXECUTE_READWRITE, &oldPrection);
```

#BHEU @BLACK HAT EVENTS

(Taken from BlackHat 2019, an hiding technique from plugins that check the VAD's protection, like MalFind)

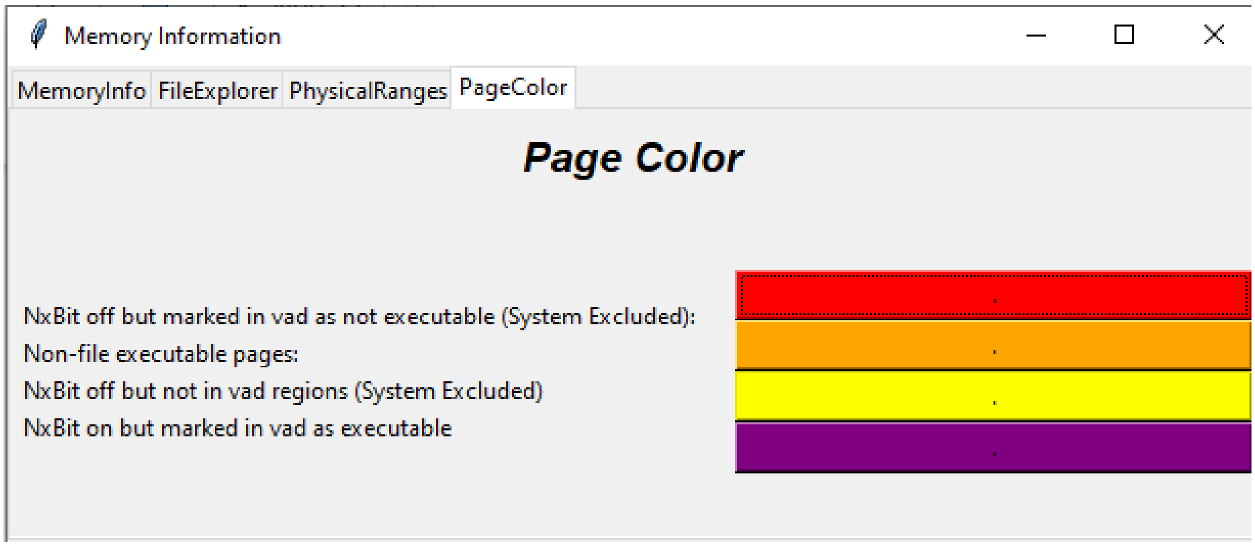
RAMMap will catch this technique and color it in red (if NxBit is off but the VAD shows it has no execute rights). For the opposite (NxBit is set but VAD shows it has execute rights), it will color in purple by default.

During the research I have found that on some Windows versions the OS will map a file with a PAGE_EXECUTE_WRITECOPY permission, but set the NxBit (so Windows will unset it manually when the CPU throw an exception).

This is acceptable because when DEP is disabled and the CPU throws an exception that a code attempted to run from a non-executable page (because the NxBit is on) - the OS handles the exception by unsetting the NxBit, so the processor could run this - which could indicate malicious injection into a process where DEP is disabled.

Abilities

You can always change the default colors by using the PageColor tab (you can also disable the coloring by changing the color to white like the other tabs)



- Color pages as you can see in the picture above
- Find if page is a part of a file (and VAD protection).
- Find the process that loaded a physical page (if it is a mapped file, then it will find all the processes that mapped this page).
- Get additional information like how the system uses this page (Use tab), the page priority, NxBit state and much more (as you can see in the picture above).

WinObj.py

Contains only one plugin: WinObj

WinObj – windows (kernel) objects

Introduction

The WinObj plugin parses the entire Windows object directory.

Motivation

After I created VolExp, while working on FileScanGui I was looking for a way to translate "harddrivevolume{number}" to the drive letter, and what is a better way than the way Windows itself does it?

So I was able to do so using WinObj (https://github.com/kslgroup/WinObj), but after using it a bit, I realized that this plugin works very well, but could sometimes miss some objects from the object manager because it doesn't parse it correctly so not all the data is present.

Thus, I decided to update the plugin. I also noticed that it has a lot of exceptions in Windows 10, so I fixed that as well.

Startup Arguments:

This plugin accepts the following arguments:

- --PARSE-ALL: Parse the entire directory (Boolean)
- --SUPLY-ADDR: Parse _OBJECT_DIRECTORY at a specific address
- --FULL-PATH: Parse the specified directory from the object directory

Some Research

Firstly, I was looking for a better way to find the object directory, and since Volatility3 parses the kernel symbols as well, we should just take the ObpRootDirectoryObject symbol which holds a pointer to the root directory of the object manager (and if it failed to get it this way, which should never happen, we can try taking this pointer from the kdbg instead).

```
def get_root_directory(self):
        """
        :return          : a pointer to the root directory
        """
        # gets the pointer
        # if for some reason ObpRootDirectoryObject not exist lets take the value from ObpRootDirectoryObject
        try:
                import struct
                _pointer_struct = struct.Struct("<Q") if self.ntkrnlmp.get_type('pointer').size == 8 else struct.Struct('I')
                root_dir_addr = int(_pointer_struct.unpack(
                        self.context.layers['primary'].read(self.ntkrnlmp.get_symbol('ObpRootDirectoryObject').address + self.ntkrnlmp.offset, 8))[0])
        except:
                root_dir_addr = info.Info.get_kdbg_structure(self.context, self.config_path, self.config['primary'], self.config['nt_symbols']).ObpRootDirectoryObject
                root_dir_addr = self.ntkrnlmp.object("pointer", offset=root_dir_addr - self.ntkrnlmp.offset)
        return root_dir_addr
```

Secondly, we parse the directory as we should parse it, as an _OBJECT_DIRECTORY struct, and make sure that we don't follow the same pointer twice (could be a while true problem in broken images):

```
def parse_directory(self, addr, l):
        """
        :param addr      : long, pointer the the driectory
        :param l         : list
        :return          : None
        the function will parse the directory and add every valid object to the received list
        """
        seen = set()
        layer_name = self.config['primary']
        kvo = self.context.layers[layer_name].config["kernel_virtual_offset"]
        directory_array = self.ntkrnlmp.object('_OBJECT_DIRECTORY', addr - self.ntkrnlmp.offset)
        for pointer_addr in directory_array.HashBuckets:
                if not pointer_addr or pointer_addr == 0xffffffff:
                        continue

                # Walk the ChainLink foreach item inside the directory.
                while pointer_addr not in seen:
                        try:
                                myObj = self.ntkrnlmp.object("pointer", offset=pointer_addr+self.POINTER_SIZE - kvo)
                                self.AddToList(myObj, l)
                        except exceptions.InvalidAddressException:
                                pass

                        seen.add(pointer_addr)
                        try:
                                pointer_addr = pointer_addr.ChainLink
                        except exceptions.InvalidAddressException:
                                break
                        if not pointer_addr:
                                break
```

Abilities

- Example of listing drivers from the Driver directory inside the root directory, finding some very old malware drivers (Stuxnet):

```
c:\volE>vol.py -f C:\sample007.bin windows.winobj --FULL-PATH \Driver
Volatility 3 Framework 1.2.1-beta.1
Progress:    0.00               Scanning primary2 using PdbSignatureScanner
Object Address(V)       Name    str     Additional Info

0x820d7418      Beep    Driver  Full Name: \Driver\Beep
0x81ea1ca0      NDIS    Driver  Full Name: \Driver\NDIS
0x8239a240      KSecDD  Driver  Full Name: \Driver\KSecDD
0x8210cf38      Raspti  Driver  Full Name: \Driver\Raspti
0x81ed39e8      es1371  Driver  Full Name: \Driver\es1371
0x8210cdc0      Mouclass        Driver  Full Name: \Driver\Mouclass
0x8232f4d0      vmx_svga        Driver  Full Name: \Driver\vmx_svga
0x82077dc8      Fips    Driver  Full Name: \Driver\Fips
0x82282b00      Kbdclass        Driver  Full Name: \Driver\Kbdclass
0x82079410      VgaSave Driver  Full Name: \Driver\VgaSave
0x822a9750      NDProxy Driver  Full Name: \Driver\NDProxy
0x8229db98      Compbatt        Driver  Full Name: \Driver\Compbatt
0x81edd030      Ptilink Driver  Full Name: \Driver\Ptilink
0x822ec8c0      MountMgr        Driver  Full Name: \Driver\MountMgr
0x81da6f38      wdmaud  Driver  Full Name: \Driver\wdmaud
0x822a3168      dmload  Driver  Full Name: \Driver\dmload
0x823c8278      isapnp  Driver  Full Name: \Driver\isapnp
0x8208a030      redbook Driver  Full Name: \Driver\redbook
0x82282150      vmmouse Driver  Full Name: \Driver\vmmouse
0x82320308      atapi   Driver  Full Name: \Driver\atapi
0x81ea1a08      vmscsi  Driver  Full Name: \Driver\vmscsi
0x82061880      RasAcd  Driver  Full Name: \Driver\RasAcd
0x81ebcea0      PSched  Driver  Full Name: \Driver\PSched
0x8226c710      dmio    Driver  Full Name: \Driver\dmio
0x820786e8      NPF     Driver  Full Name: \Driver\NPF
0x821a12c0      IpNat   Driver  Full Name: \Driver\IpNat
0x82214c68      audstub Driver  Full Name: \Driver\audstub
0x81ee1c28      usbuhci Driver  Full Name: \Driver\usbuhci
0x81d9c978      mouhid  Driver  Full Name: \Driver\mouhid
0x821a2b10      Win32k  Driver  Full Name: \Driver\Win32k
0x82305a68      swenum  Driver  Full Name: \Driver\swenum
0x8208b798      rdpdr   Driver  Full Name: \Driver\rdpdr
0x82085f38      usbhub  Driver  Full Name: \Driver\usbhub
0x821a1458      RDPCDD  Driver  Full Name: \Driver\RDPCDD
0x822f3e48      Update  Driver  Full Name: \Driver\Update
0x82277850      RasPppoe        Driver  Full Name: \Driver\RasPppoe
0x81ed3da0      usbccgp Driver  Full Name: \Driver\usbccgp
0x81e7cc08      HTTP    Driver  Full Name: \Driver\HTTP
0x82305f38      TermDD  Driver  Full Name: \Driver\TermDD
0x8233cee8      Ftdisk  Driver  Full Name: \Driver\Ftdisk
0x8205ab10      sysaudio        Driver  Full Name: \Driver\sysaudio
0x82214518      Rasl2tp Driver  Full Name: \Driver\Rasl2tp
0x8208b0d0      Fdc     Driver  Full Name: \Driver\Fdc
0x82325030      ParVdm  Driver  Full Name: \Driver\ParVdm
0x8232fb60      vmci    Driver  Full Name: \Driver\vmci
0x81e816c8      PptpMiniport    Driver  Full Name: \Driver\PptpMiniport
0x81ee15b0      vmxnet  Driver  Full Name: \Driver\vmxnet
0x8208b620      serenum Driver  Full Name: \Driver\serenum
0x823afce8      WMIxWDM Driver  Full Name: \Driver\WMIxWDM
0x823e8160      ACPI_HAL        Driver  Full Name: \Driver\ACPI_HAL
0x81f26870      MRxCls  Driver  Full Name: \Driver\MRxCls
0x820e54f8      MRxNet  Driver  Full Name: \Driver\MRxNet
0x820dba18      NetBT   Driver  Full Name: \Driver\NetBT
0x8226c0a8      agp440  Driver  Full Name: \Driver\agp440
0x823058d8      Cdrom   Driver  Full Name: \Driver\Cdrom
```

- Example of results in vol3 - 1.2.1 (Windows XP) for driverscan and drivers (so the only way to get correct results at the moment is my way):

```
c:\volE>vol.py -f C:\sample007.bin windows.driverscan
Volatility 3 Framework 1.2.1-beta.1
Progress:    0.00               Scanning primary2 using PdbSignatureScanner
Offset  Start   Size    Service Key     Driver Name     Name


c:\volE>_
```

```
c:\volE>vol.py -f C:\sample007.bin windows.drivers
Volatility 3 Framework 1.2.1-beta.1
Progress:    0.00               Scanning primary2 using PdbSignatureScanner
Offset  Start   Size    Service Key     Driver Name     Name


c:\volE>
```

- Using WinObj, you can enumerate everything from the object directory itself:
  As we saw already, you can enumerate drivers, but there is much more you can do. You can enumerate devices, callbacks, mutexes, events, semaphores, jobs, ALPC ports (including the WSL ALPC ports, from \PSXSS), symbolic links, section objects, windows stations, sessions, desktops, etc..
  You can recreate almost any plugin results from WinObj:
  For example wintree (from volatility2) -> using WinObj to parse \Windows\WindowStations and find all the windowstations objects, then parse all the processes from that window, which will show all the information that wintree gives us.
- The kernel uses the object directory to translate objects as well so it could be very unsafe for a malware to change data there and try to hide, because it could result in unwanted changes.

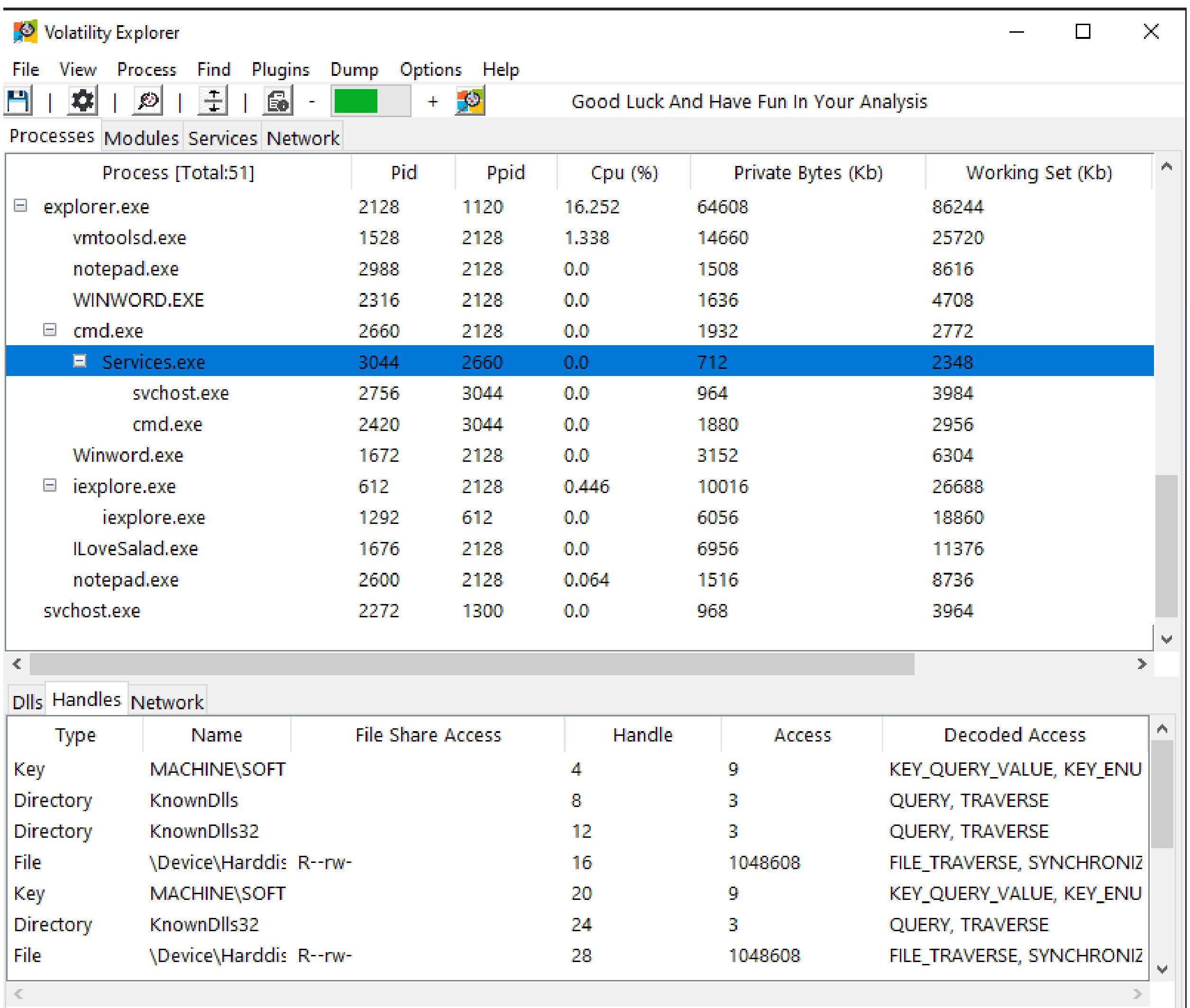
Using the tools

Instructions

In this demo I am going to use two different dumps. One that has been analyzed before (https://drive.google.com/file/d/0B7v1Owo0v5SYZ016VmVoVFV1elE/view), the KSLSample.vmem - the sample of the threadmap plugin in the contest of 2017, as well as sahar.vmem, which I am not allowed to upload.

Sahar.vmem

This memory image belongs to a cloud VMWare user that downloaded a malicious program.

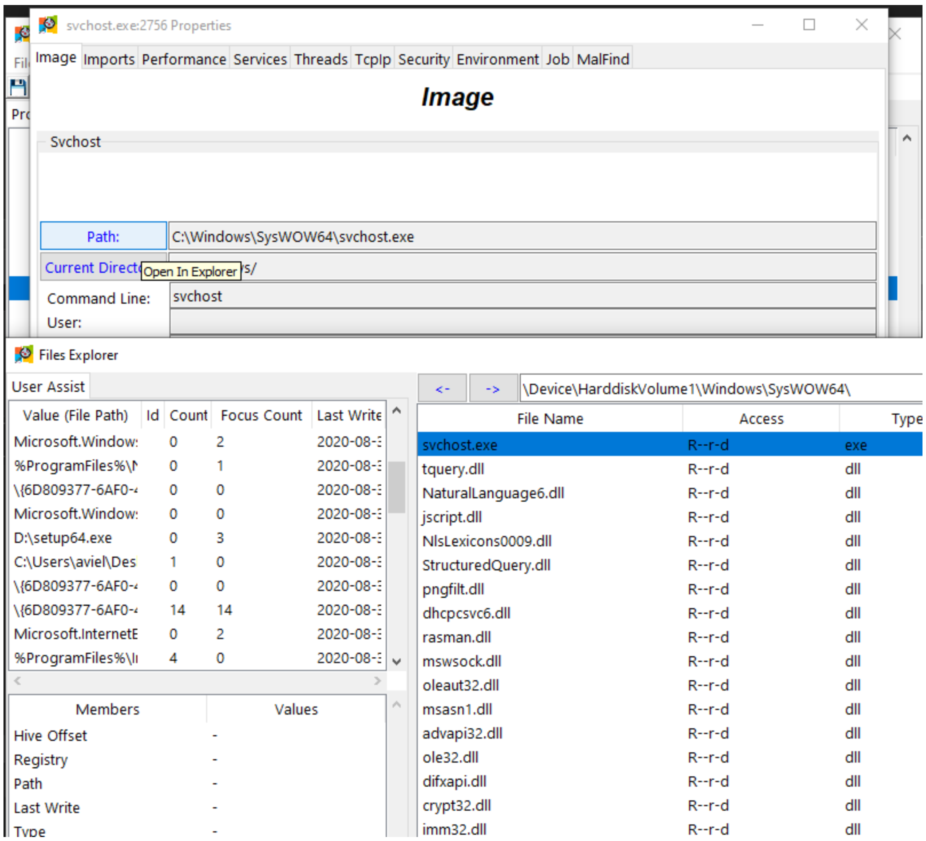


From the process tree already, we can see a lot of weird stuff going on here.

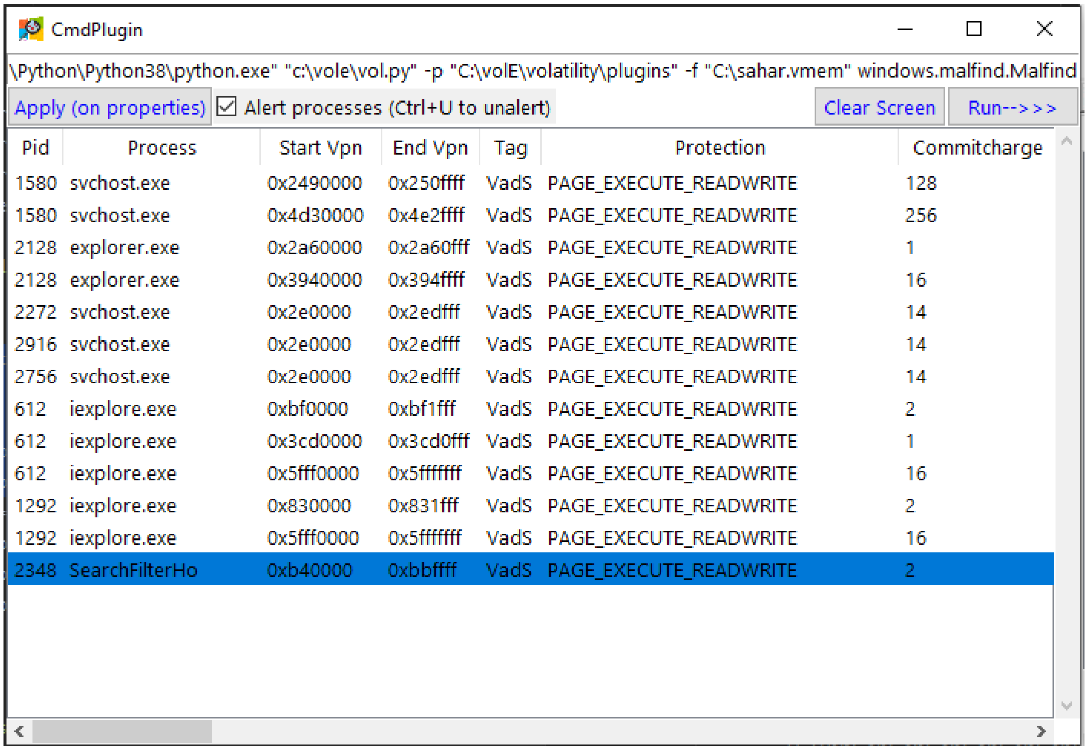
Services.exe is running under a cmd, svchost is running under Services, two instances of differently spelled WinWord variants (lower and upper case).

Opening svchost.exe's (2756) properties and clicking at Path to open in Explorer validated that we are in fact dealing with the authentic svchost image.

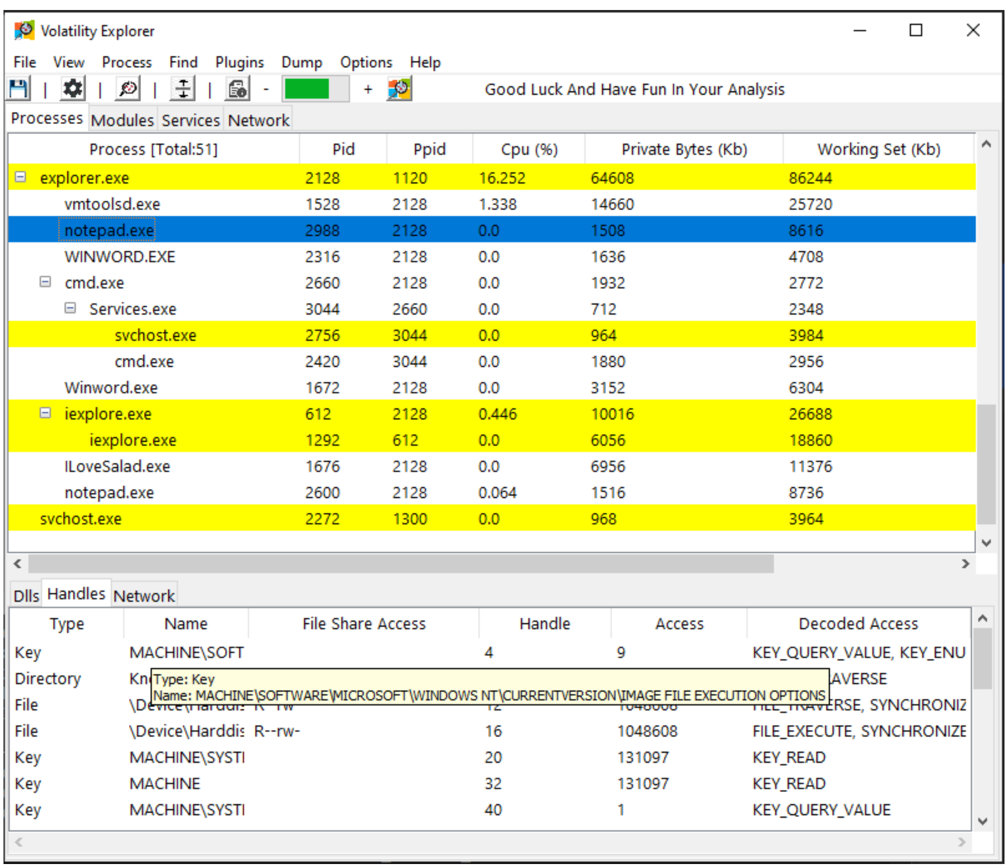
In that case, what is going on here??



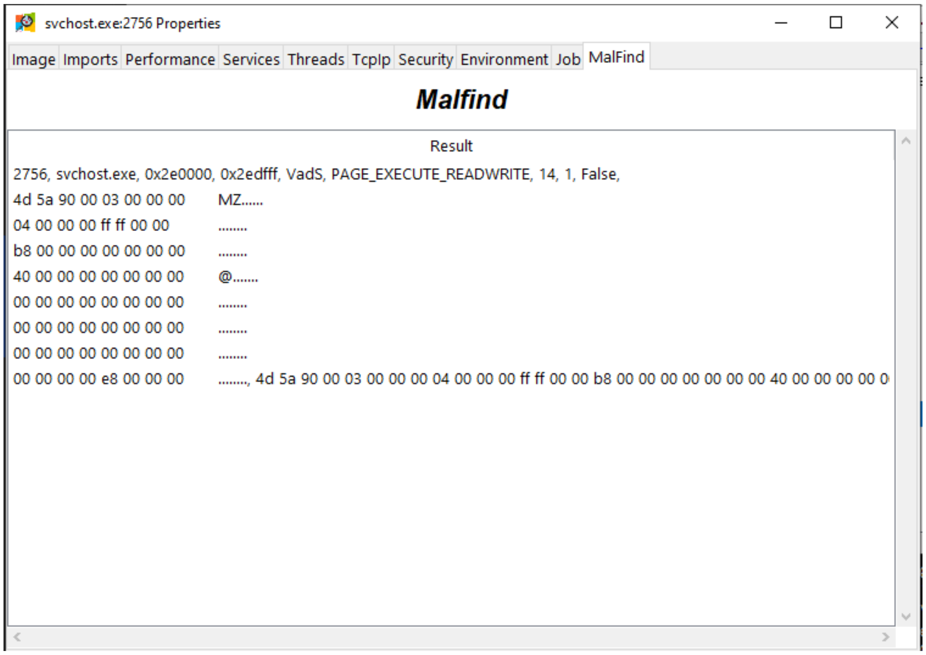
To make sure, Let's run Malfind against this image (you can run Malfind on one specific process by using Process->Plugins->WellKnown->Malfind)



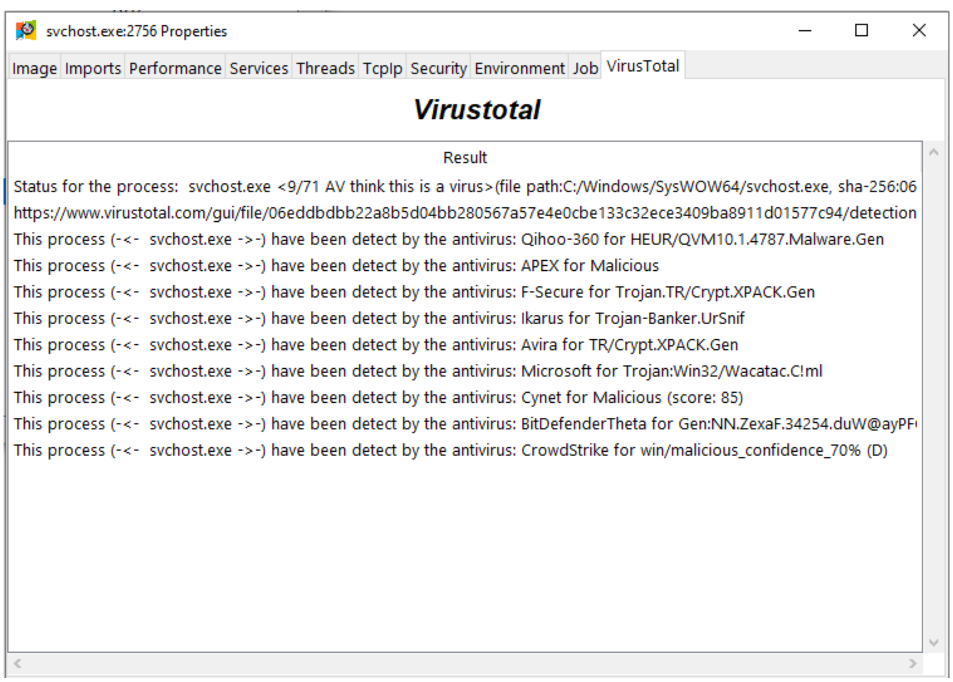
Using the Apply (on properties) button will color the svchost in yellow:



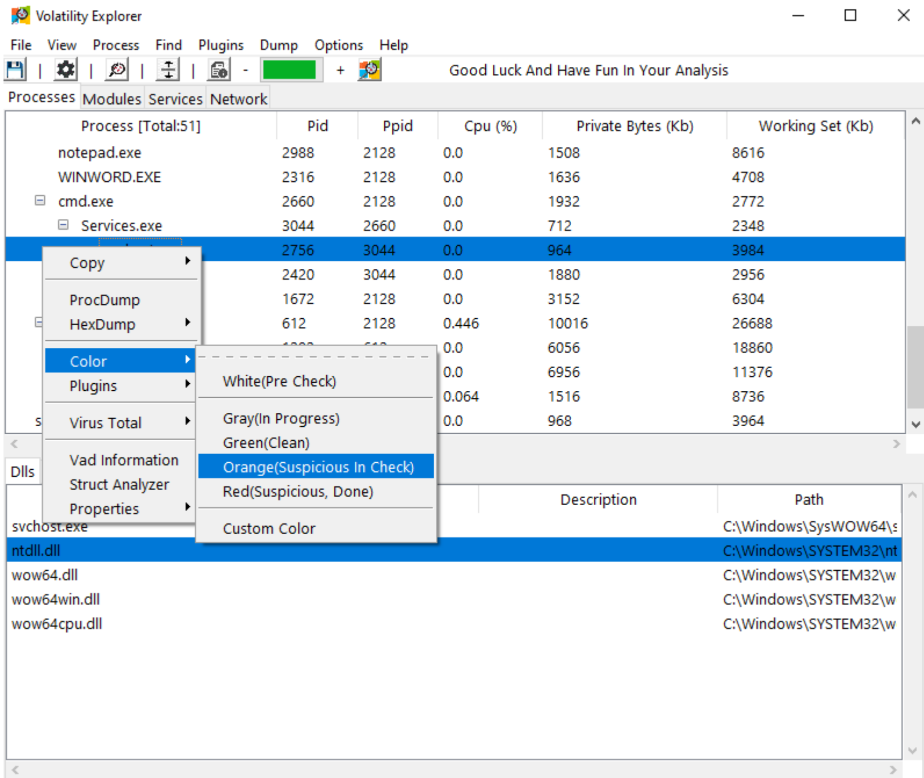
By looking at svchost's properties we can understand that we've got an MZ injection in our hands.



By uploading this process to VirusTotal we can see that nine different antiviruses detected this file as malicious (none of them detected the real svchost):

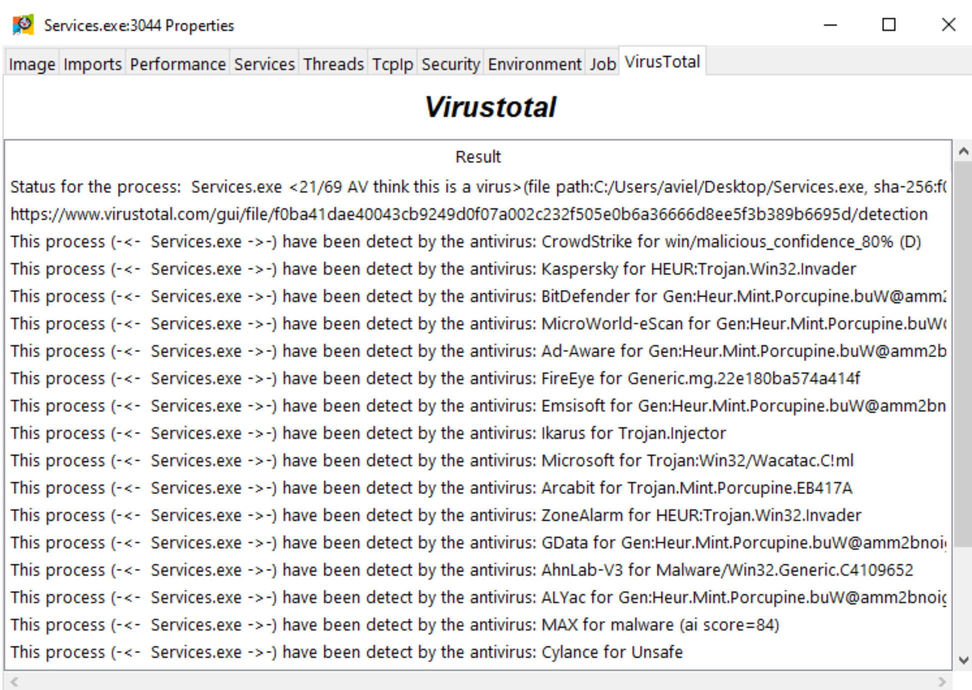


Let's add a comment on this in the Comment tab and color this process as suspicious



As we can see, this svchost loads only 4 dlls. Coincidentally, services.exe loads the same dlls exactly.

In that case, let's take a look at Services.exe:



A quick view of the process' imports, and we can already know what we are dealing with:

Services.exe:3044 Properties

Image | Imports | Performance | Services | Threads | TcpIp | Security | Environment | Job | VirusTotal

Imports

Function	Module	Address	Hint
LoadLibraryA	KERNEL32.dll	0xa42000	0x2f1
GetProcAddress	KERNEL32.dll	0xa42004	0x220
ReadProcessMemory	KERNEL32.dll	0xa42008	0x36b
CreateProcessA	KERNEL32.dll	0xa4200c	0x94
CreateFileA	KERNEL32.dll	0xa42010	0x78
GetFileSize	KERNEL32.dll	0xa42014	0x1d4
ReadFile	KERNEL32.dll	0xa42018	0x368
GetModuleHandleA	KERNEL32.dll	0xa4201c	0x1f6
VirtualAllocEx	KERNEL32.dll	0xa42020	0x455
WriteProcessMemory	KERNEL32.dll	0xa42024	0x496
GetThreadContext	KERNEL32.dll	0xa42028	0x25c
SetThreadContext	KERNEL32.dll	0xa4202c	0x406
ResumeThread	KERNEL32.dll	0xa42030	0x38d
GetModuleFileNameA	KERNEL32.dll	0xa42034	0x1f4
GetCurrentProcessId	KERNEL32.dll	0xa42038	0x1aa
GetCurrentThreadId	KERNEL32.dll	0xa4203c	0x1ad
GetTickCount	KERNEL32.dll	0xa42040	0x266
QueryPerformanceCounter	KERNEL32.dll	0xa42044	0x354

CreateProcessA, CreateFileA, GetFileSize, ReadFile, VirtualAllocEx, WriteProcessMemory, Get/SetThreadContext and ResumeThread is almost everything we need to perform a process hollowing injection.

If we take a look at the VAD information (using process Right Click -> VAD information):

Services.exe (3044): Memory information (from VAD)

Start	Size	Use	Pr	Control Flags
0x7ffe0000	0xffff	User Shared Data	PA	
0xa40000	0x5fff	\Users[redacted]\Desktop\Services.exe	PA	Accessed, File, Image
0x76ad0000	0x45fff	\Windows\SysWOW64\KernelBase.dll	PA	File, Image
0x74540000	0x4bfff	\Windows\SysWOW64\apphelp.dll	PA	Accessed, File, Image
0x76d30000	0x10ffff	\Windows\SysWOW64\kernel32.dll	PA	Accessed, File, Image
0x775f0000	0x17ffff	\Windows\SysWOW64\ntdll.dll	PA	Accessed, File, Image
0x40000	0xfff	\Windows\System32\apisetschema.dll	PA	File, Image
0x430000	0x66fff	\Windows\System32\locale.nls	PA	Accessed, File
0x77410000	0x1a8fff	\Windows\System32\ntdll.dll	PA	Accessed, File, Image
0x746a0000	0x3efff	\Windows\System32\wow64.dll	PA	Accessed, File, Image

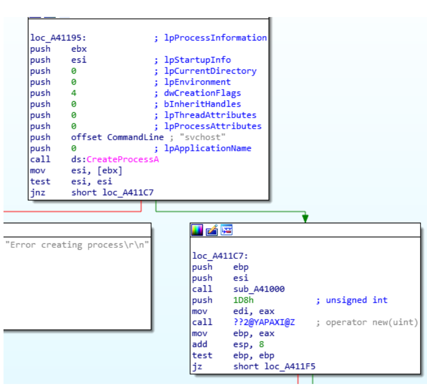
Members	Values
Start	0xa40000
End	0xa45fff
Size	0x5fff
Tag	Vad
Flags	'Protection': 7, 'VadType': 2
Protection	PAGE_EXECUTE_WRITECOPY
Type	2
Control Area	0xfa800348a010
Segment	0xfa800348a010

We can see that this process was loaded from the desktop, and also loads ntdll.dll dynamically (using the LoadLibraryA and GetProcAddress that we saw in the imports above).

Hah!, NtUnmapViewOfSection (from ntdll) is the only missing function to perform process hollowing.

We can also dump Services.exe and view it in our favorite disassembly tool to confirm.

Here it is creating a process named svchost:



```
mov     edi, [esp+48h+lpFileName]
add     esp, 0Ch
push    0               ; hTemplateFile
push    0               ; dwFlagsAndAttributes
push    4               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    0               ; dwShareMode
push    80000000h       ; dwDesiredAccess
push    edi             ; lpFileName
call    ds:CreateFileA
mov     ebp, eax
cmp     ebp, 0FFFFFFFFh
jnz     short loc_A41261
```

Here it opens the malicious file that will be used to replace svchost:

```
push    offset ModuleName ; "ntdll"
call    ds:GetModuleHandleA
push    offset aNtunmapviewofs ; "NtUnmapViewOfSection"
push    eax             ; hModule
call    ds:GetProcAddress
mov     ecx, [esi+8]
mov     edx, [ebx]
push    ecx
push    edx
call    eax
test    eax, eax
jz      short loc_A412F3
```

It then unmaps svchost’s sections:

```
mov     eax, [ebp+50h]
mov     ecx, [esi+8]
mov     edx, [ebx]
add     esp, 4
push    40h             ; flProtect
push    3000h           ; flAllocationType
push    eax             ; dwSize
push    ecx             ; lpAddress
push    edx             ; hProcess
call    ds:VirtualAllocEx
test    eax, eax
jnz     short loc_A41333
```

It continues to allocate additional memory:

```
mov     ecx, [esp+48h+Buffer]
mov     edx, [ecx+18h]
mov     ecx, [edi+edx+10h]
lea     eax, [edi+edx]
mov     edx, [eax+14h]
add     edx, [esp+48h+var_1C]
mov     eax, [ebx]
add     esp, 0Ch
push    0               ; lpNumberOfBytesWritten
push    ecx             ; nSize
push    edx             ; lpBuffer
push    ebp             ; lpBaseAddress
push    eax             ; hProcess
call    ds:WriteProcessMemory
test    eax, eax
jz      short loc_A4138A
```

Rewrite the headers:

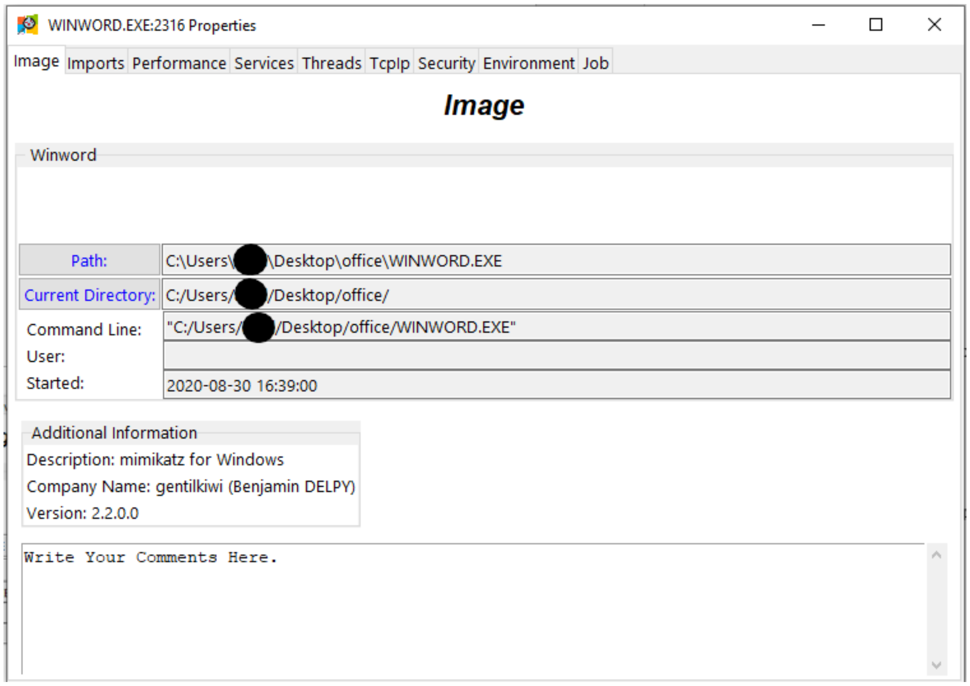
```
mov     edx, [ebx+4]
add     esp, 4
push    esi             ; lpContext
push    edx             ; hThread
call    ds:GetThreadContext
test    eax, eax
jnz     short loc_A41626
```

Gets thread context:

```
push    ecx             ; hThread
call    ds:ResumeThread
test    eax, eax
jnz     short loc A41681
```

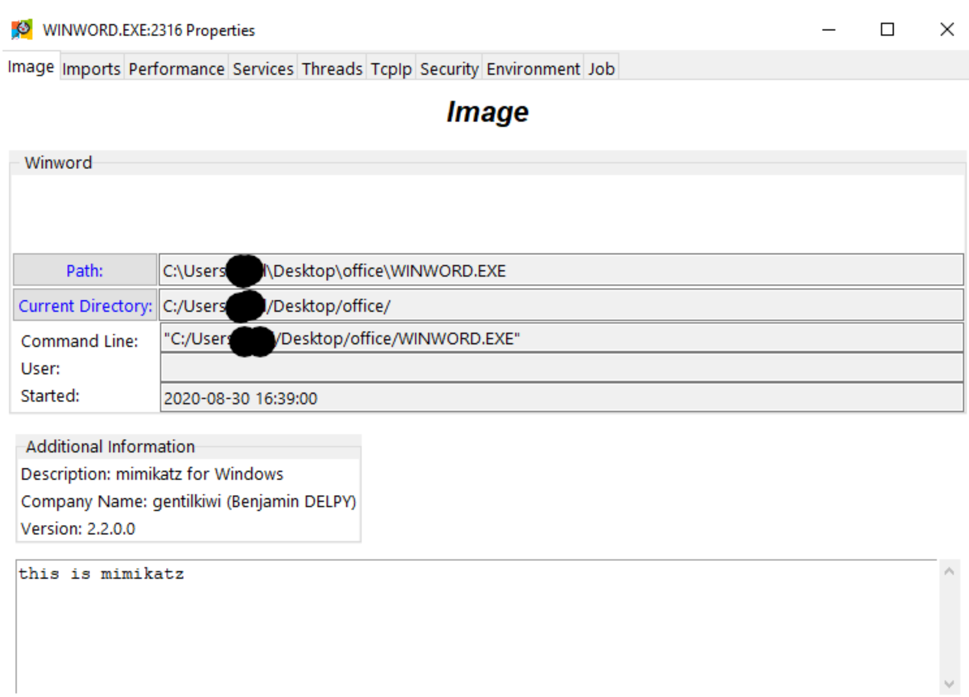
It then resumes the thread, and that's it.

If we also disassemble the svchost, we can find out that it does a lot of abnormal stuff as well, such as running one of our WinWord.exes. Let’s look at it:



And just by viewing the file description we can immediately see that this is a mimikatz.

Adding a comment to make our analysis easier:



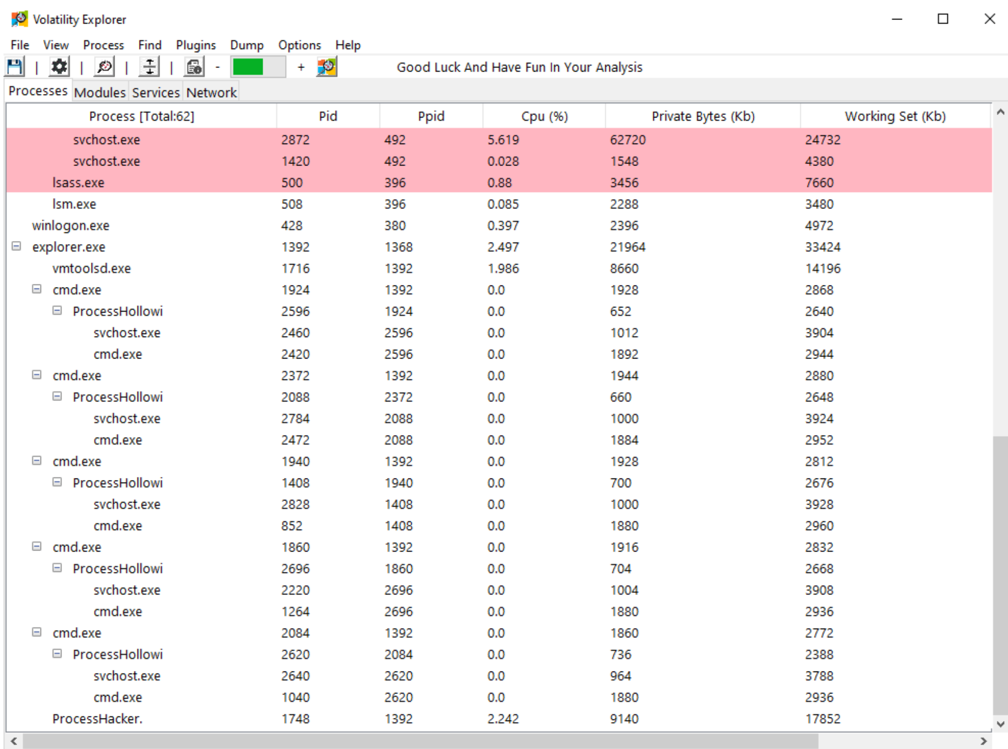
KSLSample.vmem

This is the dump used by kslgroup who created the threadmap.py plugin:

https://drive.google.com/file/d/0B7v1Owo0v5SYZ016VmVoVFV1elE/view

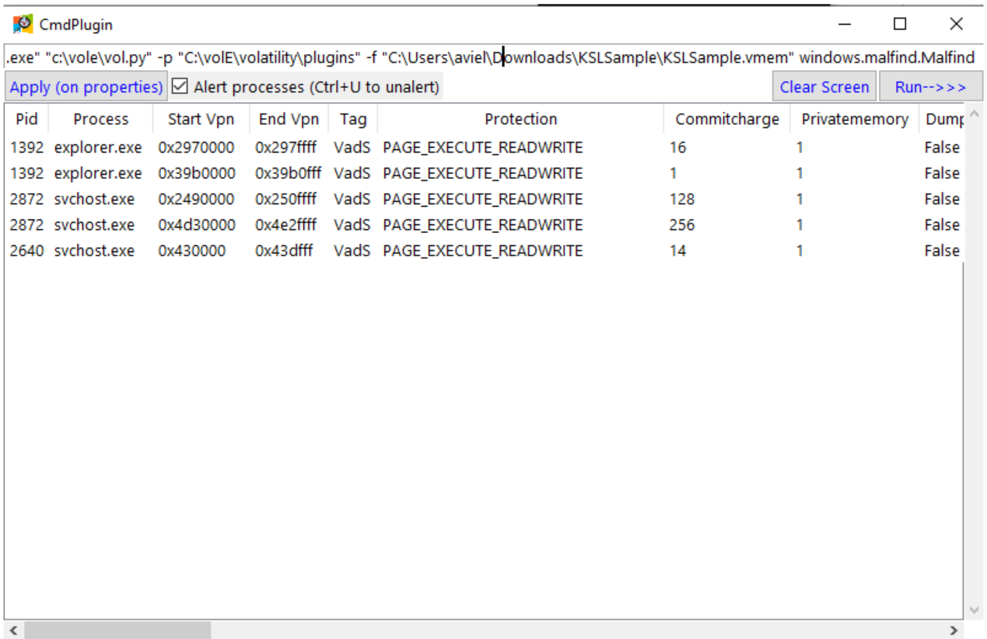
Here, the process hollowing is trying to hide itself by allocating the memory as PAGE_READWRITE and changing it later to be executeable.

Let's take a look in VolExp:



Process [Total:62]	Pid	Ppid	Cpu (%)	Private Bytes (Kb)	Working Set (Kb)
svchost.exe	2872	492	5.619	62720	24732
svchost.exe	1420	492	0.028	1548	4380
lsass.exe	500	396	0.88	3456	7660
lsm.exe	508	396	0.085	2288	3480
winlogon.exe	428	380	0.397	2396	4972
explorer.exe	1392	1368	2.497	21964	33424
vmtoolsd.exe	1716	1392	1.986	8660	14196
cmd.exe	1924	1392	0.0	1928	2868
ProcessHollowi	2596	1924	0.0	652	2640
svchost.exe	2460	2596	0.0	1012	3904
cmd.exe	2420	2596	0.0	1892	2944
cmd.exe	2372	1392	0.0	1944	2880
ProcessHollowi	2088	2372	0.0	660	2648
svchost.exe	2784	2088	0.0	1000	3924
cmd.exe	2472	2088	0.0	1884	2952
cmd.exe	1940	1392	0.0	1928	2812
ProcessHollowi	1408	1940	0.0	700	2676
svchost.exe	2828	1408	0.0	1000	3928
cmd.exe	852	1408	0.0	1880	2960
cmd.exe	1860	1392	0.0	1916	2832
ProcessHollowi	2696	1860	0.0	704	2668
svchost.exe	2220	2696	0.0	1004	3908
cmd.exe	1264	2696	0.0	1880	2936
cmd.exe	2084	1392	0.0	1860	2772
ProcessHollowi	2620	2084	0.0	736	2388
svchost.exe	2640	2620	0.0	964	3788
cmd.exe	1040	2620	0.0	1880	2936
ProcessHacker.	1748	1392	2.242	9140	17852

This is just a research dump, so they did not make any special effort to try and hide the process hollowing. But can Malfind find it?

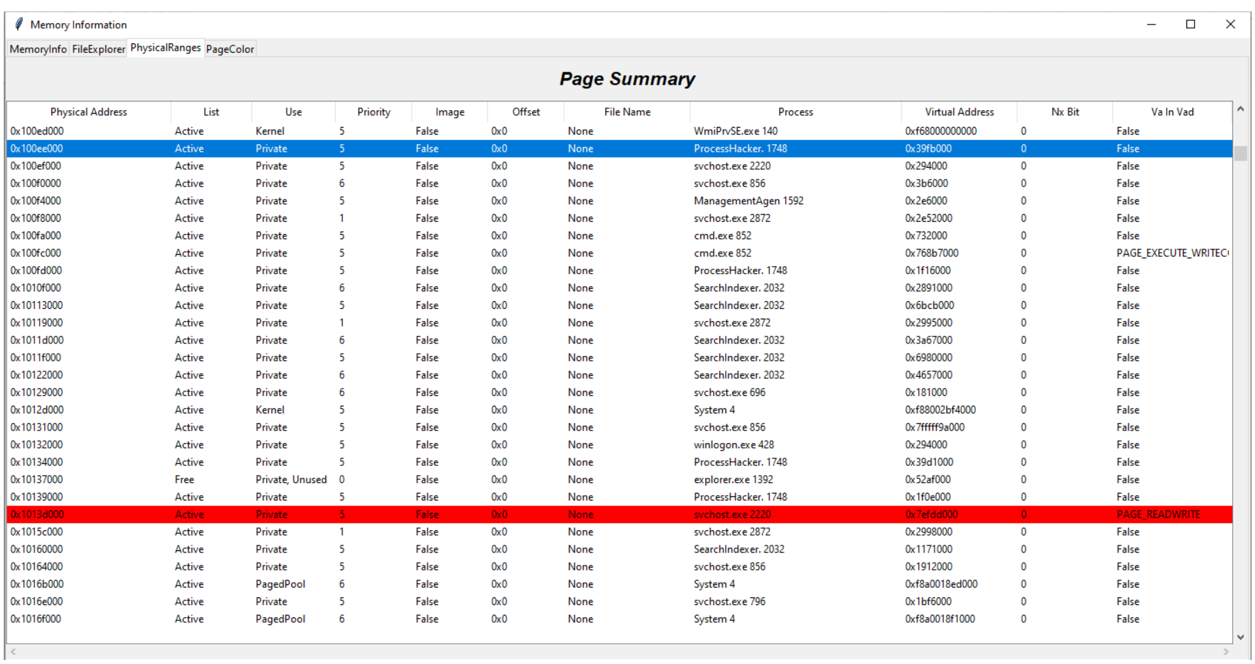


Pid	Process	Start Vpn	End Vpn	Tag	Protection	Commitcharge	Privatememory	Dump
1392	explorer.exe	0x2970000	0x297ffff	VadS	PAGE_EXECUTE_READWRITE	16	1	False
1392	explorer.exe	0x39b0000	0x39b0fff	VadS	PAGE_EXECUTE_READWRITE	1	1	False
2872	svchost.exe	0x2490000	0x250ffff	VadS	PAGE_EXECUTE_READWRITE	128	1	False
2872	svchost.exe	0x4d30000	0x4e2ffff	VadS	PAGE_EXECUTE_READWRITE	256	1	False
2640	svchost.exe	0x430000	0x43dfff	VadS	PAGE_EXECUTE_READWRITE	14	1	False

As we can see, Malfind only found that process 2640 is suspicious because it has started with the PAGE_EXECUTE_READWRITE rights. They probably did that as a way of comparison.

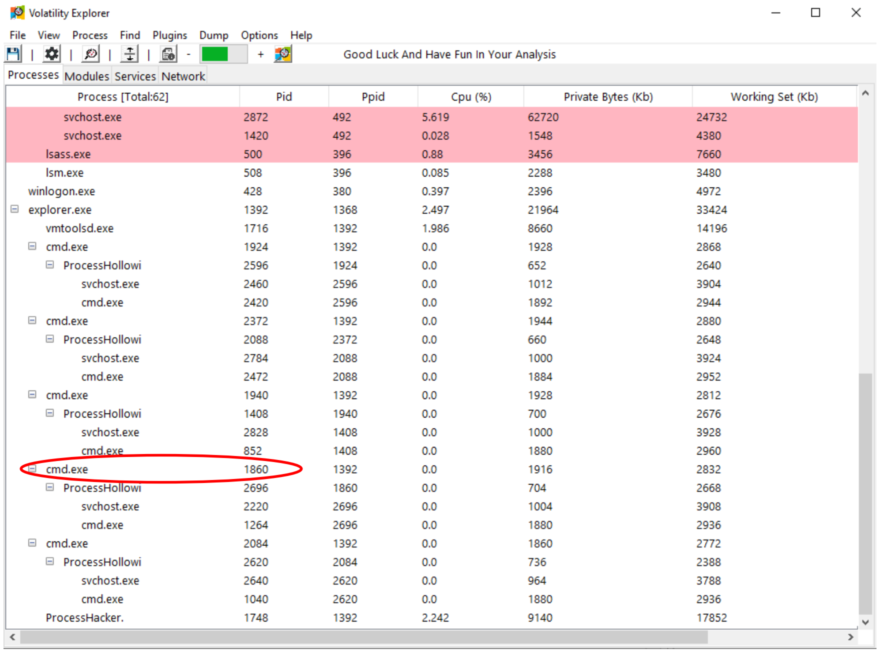
Let's see now how RAMMap can catch this as well.

Firstly, the plugin won't mark 2640 as suspicious because it has both the NXBit off and has Execute rights. RAMMap only marks pages with a weird state, like a page where the VAD protection is Non executable but the NxBit is not set (so the page can execute):



Physical Address	List	Use	Priority	Image	Offset	File Name	Process	Virtual Address	Nx Bit	Va In Vad
0x100ed000	Active	Kernel	5	False	0x0	None	WmiPrvSE.exe 140	0xf68000000000	0	False
0x100ee000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x39fb000	0	False
0x100ef000	Active	Private	5	False	0x0	None	svchost.exe 2220	0x294000	0	False
0x100f0000	Active	Private	6	False	0x0	None	svchost.exe 856	0x3b6000	0	False
0x100f4000	Active	Private	5	False	0x0	None	ManagementAgen 1592	0x2e6000	0	False
0x100f8000	Active	Private	1	False	0x0	None	svchost.exe 2872	0x2e52000	0	False
0x100fa000	Active	Private	5	False	0x0	None	cmd.exe 852	0x732000	0	False
0x100fc000	Active	Private	5	False	0x0	None	cmd.exe 852	0x768b7000	0	PAGE_EXECUTE_WRITEC
0x100fd000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x1f16000	0	False
0x1010f000	Active	Private	6	False	0x0	None	SearchIndexer. 2032	0x2891000	0	False
0x10113000	Active	Private	5	False	0x0	None	SearchIndexer. 2032	0x6bcb000	0	False
0x10119000	Active	Private	1	False	0x0	None	svchost.exe 2872	0x2995000	0	False
0x1011d000	Active	Private	6	False	0x0	None	SearchIndexer. 2032	0x3a67000	0	False
0x1011f000	Active	Private	5	False	0x0	None	SearchIndexer. 2032	0x6980000	0	False
0x10122000	Active	Private	6	False	0x0	None	SearchIndexer. 2032	0x4657000	0	False
0x10129000	Active	Private	6	False	0x0	None	svchost.exe 696	0x181000	0	False
0x1012d000	Active	Kernel	5	False	0x0	None	System 4	0xf88002bf4000	0	False
0x10131000	Active	Private	5	False	0x0	None	svchost.exe 856	0x7fffff9a000	0	False
0x10132000	Active	Private	5	False	0x0	None	winlogon.exe 428	0x294000	0	False
0x10134000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x39d1000	0	False
0x10137000	Free	Private, Unused	0	False	0x0	None	explorer.exe 1392	0x52af000	0	False
0x10139000	Active	Private	5	False	0x0	None	ProcessHacker. 1748	0x1f0e000	0	False
0x1013d000	Active	Private	5	False	0x0	None	svchost.exe 2220	0x7efdd000	0	PAGE_READWRITE
0x1015c000	Active	Private	1	False	0x0	None	svchost.exe 2872	0x2998000	0	False
0x10160000	Active	Private	5	False	0x0	None	SearchIndexer. 2032	0x1171000	0	False
0x10164000	Active	Private	5	False	0x0	None	svchost.exe 856	0x1912000	0	False
0x1016b000	Active	PagedPool	6	False	0x0	None	System 4	0xf8a0018ed000	0	False
0x1016e000	Active	Private	5	False	0x0	None	svchost.exe 796	0x1bf6000	0	False
0x1016f000	Active	PagedPool	6	False	0x0	None	System 4	0xf8a0018f1000	0	False

Here, we can see the PAGE_READWRITE as we expected, but the NxBit is unset, so this page can execute, meaning it is trying to hide, thus marking it in red.


Volatility Explorer

File View Process Find Plugins Dump Options Help

Good Luck And Have Fun In Your Analysis

Processes Modules Services Network

Process [Total:62]	Pid	Ppid	Cpu (%)	Private Bytes (Kb)	Working Set (Kb)
svchost.exe	2872	492	5.619	62720	24732
svchost.exe	1420	492	0.028	1548	4380
lsass.exe	500	396	0.88	3456	7660
lsm.exe	508	396	0.085	2288	3480
winlogon.exe	428	380	0.397	2396	4972
explorer.exe	1392	1368	2.497	21964	33424
vmtoolsd.exe	1716	1392	1.986	8660	14196
cmd.exe	1924	1392	0.0	1928	2868
ProcessHollowi	2596	1924	0.0	652	2640
svchost.exe	2460	2596	0.0	1012	3904
cmd.exe	2420	2596	0.0	1892	2944
cmd.exe	2372	1392	0.0	1944	2880
ProcessHollowi	2088	2372	0.0	660	2648
svchost.exe	2784	2088	0.0	1000	3924
cmd.exe	2472	2088	0.0	1884	2952
cmd.exe	1940	1392	0.0	1928	2812
ProcessHollowi	1408	1940	0.0	700	2676
svchost.exe	2828	1408	0.0	1000	3928
cmd.exe	852	1408	0.0	1880	2960
cmd.exe	1860	1392	0.0	1916	2832
ProcessHollowi	2696	1860	0.0	704	2668
svchost.exe	2220	2696	0.0	1004	3908
cmd.exe	1264	2696	0.0	1880	2936
cmd.exe	2084	1392	0.0	1860	2772
ProcessHollowi	2620	2084	0.0	736	2388
svchost.exe	2640	2620	0.0	964	3788
cmd.exe	1040	2620	0.0	1880	2936
ProcessHacker.	1748	1392	2.242	9140	17852

Closing Statements

There used to be a time where the SysInternals tools were open source. You could understand the basics of how Windows manages its memory just by reading them. Those were easier times.
About 10 years ago they decided to make the SysInternals suite closed source. Since then, to understand the theoretics behind Windows you need to read, books like The Art of Memory Forensics and Windows Internals.
Then Windows 10 shows up and since a lot of things changed, Volatility starts to move to Volatility3, and Rekall is no longer supported… and I just wanted to understand how the memory manager changes in Windows 10 and the forensics aspects of it.

Therefore, I decided to create an improved open source tool of the most powerful tools from SysInternals (like Pavel Yosifovich started doing) , and added the memory analysis aspect to them.
In the beginning, I only created the VolExp suite for Volatility2 but it wasn't perfect.
My then mentor, Shachaf Atun, competed in the contest in previous years, so I decide to wait one year with the contest.
Later, you decided to make the contest for Volatility3, so I decided to convert the plugin and also add much more information to it, taking inspiration from the SysInternals tools.

This plugin was supposed to let other people understand the intricacy of the operating system's memory management, but I myself have learned a lot by making these plugins; just by running RAMMap and seeing the different colors in different OS versions makes you understand a bit. Slowly all the pieces combine together into the big puzzle, as I hope they will for others.

This last month of researching was intensive but I had a lot of fun, now I will recreate the friendships and connections that I have neglected, and back to normal.
Thanks to my lovely girlfriend for the understanding during this time.
I hope these tools will help you during your memory analysis.

References

1. Gitub: https://github.com/memoryforensics1/Vol3xp
2. VolExp for volatility 2: https://github.com/memoryforensics1/VolExp
3. Memory sample: https://drive.google.com/file/d/0B7v1Owo0v5SYZ016VmVoVFV1elE/view
4. Hale Ligh Michael, Case Andres, Levy Jamie, Walter Aaron, The Art of Memory Forensics, Wiley, 2014
5. Russinovich Mark, Solomon A. David, Ionescu Alex, Windows Internals 6th Edition, Microsoft Press, 2012
6. RAMMap: https://docs.microsoft.com/en-us/sysinternals/downloads/rammap
7. Process Explorer: https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer
8. WinObj: https://docs.microsoft.com/en-us/sysinternals/downloads/winobj
9. Threadmap plugin: https://github.com/kslgroup/threadmap
10. WinInternals tools: https://github.com/zodiacon/AllTools