v2

Threat Hunting Professional

Hunting Malware

Section 03 | Module 03

Table of Contents

MODULE 03 | HUNTING MALWARE

Learning Objectives

By the end of this module, you should have a better understanding of:

- ✓ Malware detection tools
- ✓ Malware detection techniques
- ✓ Memory hunting and analysis
- ✓ The importance of malware analysis

3.1

Introduction

3.1 Introduction

Malware is not going anywhere anytime soon. Malware authors use various tools and techniques to remain undetected for as long as possible.

We also need various tools and techniques to hunt for them.

3.1 Introduction

We are hunting for malware in various locations:

- Hiding in plain sight
- Injected into other processes
- In files (macros, for example)
- In email attachments
- In memory (known as Fileless malware)
- Etc.

3.1 Introduction

The tools presented in this module do not represent an exhaustive list by no means, but remember, you're being trained to hunt and to take a proactive approach.

This module will reveal that there are tools available to aid you in your hunts.

Detection Tools

3.2 Detection Tools

In this section, we will look at various tools that will aid us in hunting for malware in our networks.

Whether it's a Meterpreter session or a DLL injection, we should have a plethora of tools at our disposal when we're hunting for specific attack signatures.

3.2.1 Detection Tools – PE Capture

The NoVirusThanks's **PE Capture** tool captures PE files, executables, DLLs, and drivers loaded into the operating system. Any loaded executable (PE, EXE, etc.) is displayed within the GUI, and a copy is saved in the intercepted folder for further analysis.

The copied file is named as the hash value of the file. Additionally, the tool will log execution events to help you easily find a specific PE file that was previously captured. You can download the tool here.

http://www.novirusthanks.org/products/pe-capture/

3.2.1 Detection Tools – PE Capture

PE Capture is also available in a service-only version.

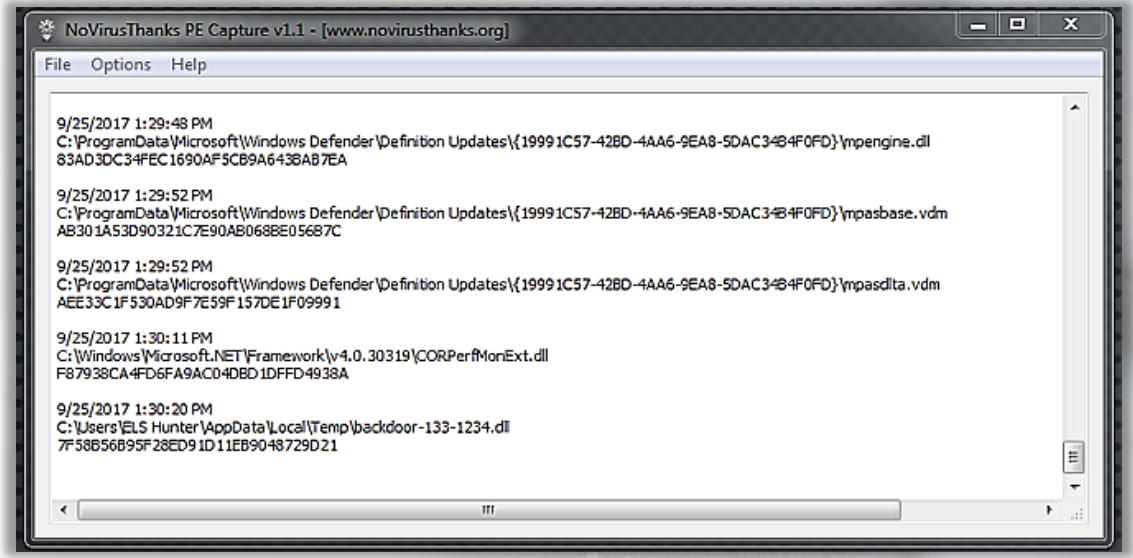
- This will allow you to install it on multiple PCs.
- It does not have a GUI.
- The program is free for personal use only. You can read more about the tool, and/or download the tool here.

http://www.novirusthanks.org/products/pe-capture-service/

3.2.1 Detection Tools – PE Capture

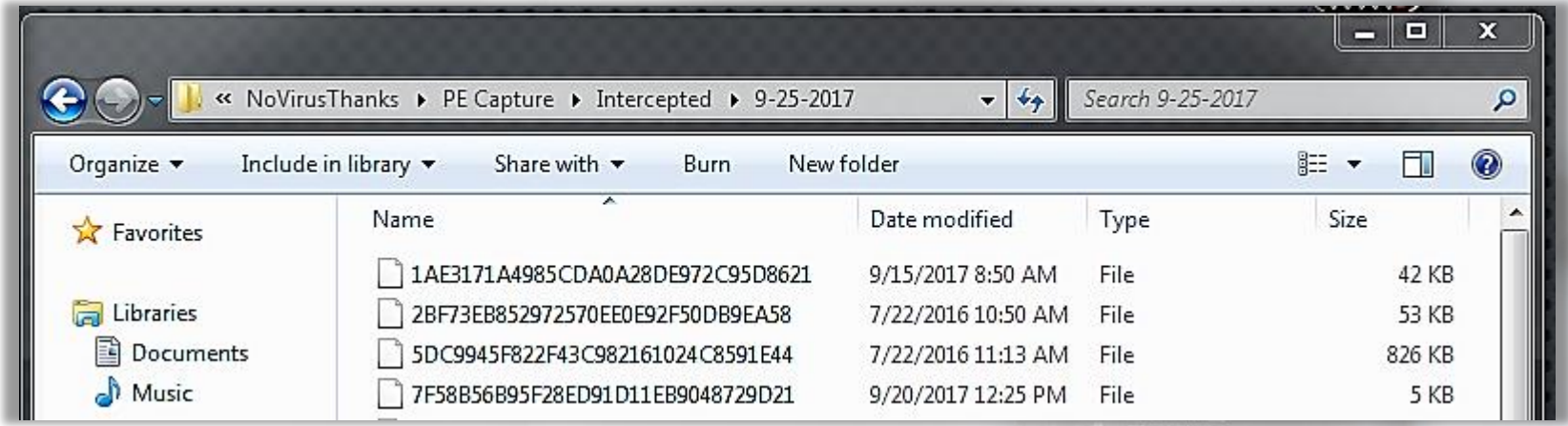
Viewing the screenshot on the right, we can see the suspicious DLL loaded in memory. The GUI shows us the path of the DLL as well as the hash.

We can now look into the Intercepted folder, or the Logs folder, to see what information is saved for us.



3.2.1 Detection Tools – PE Capture

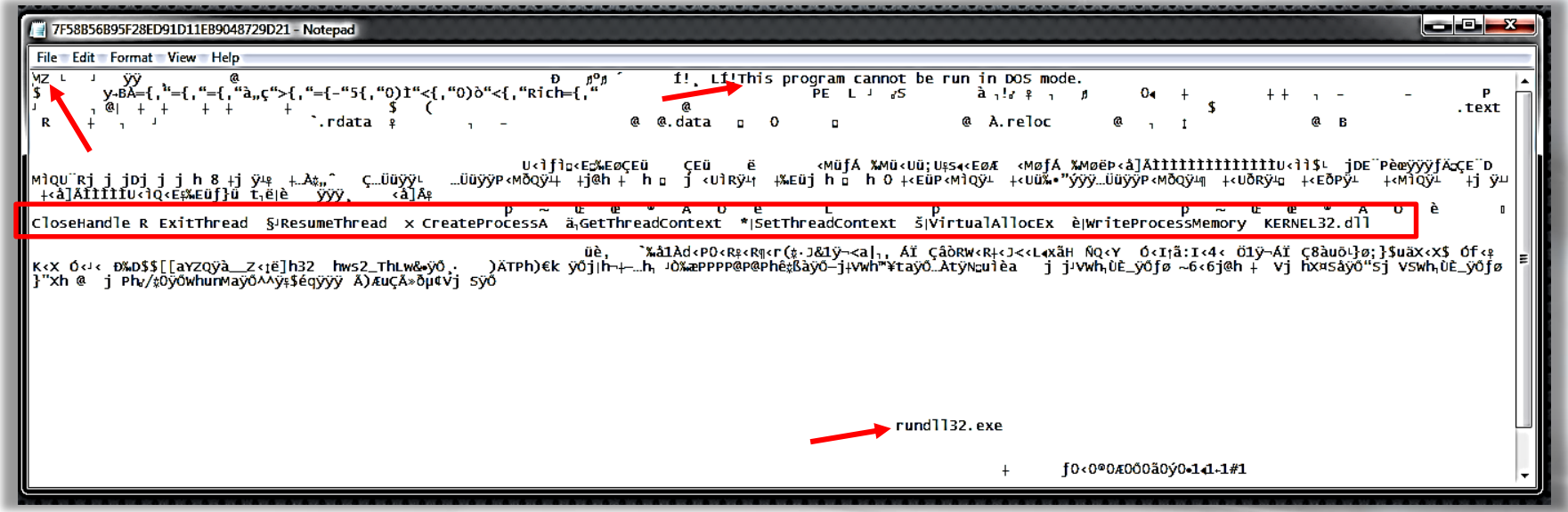
In the File menu, we can either choose **Open "Intercepted" Folder** or **Open "Logs" Folder**.



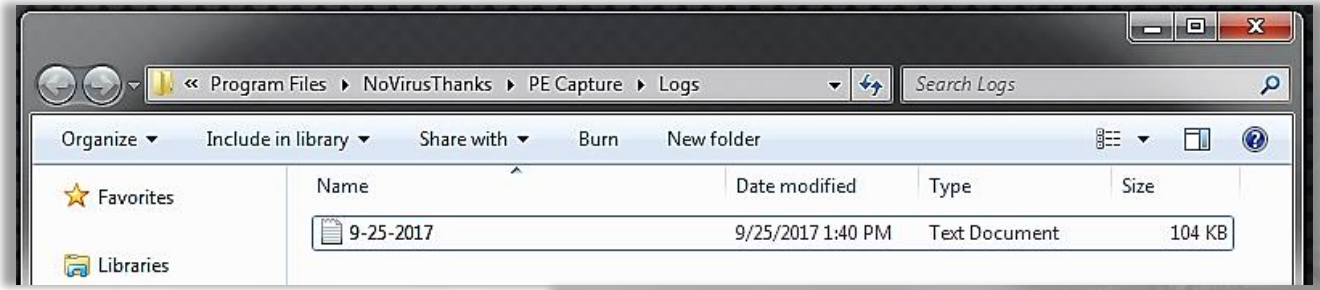
You can now analyze the exported file to see if it is benign or malicious.

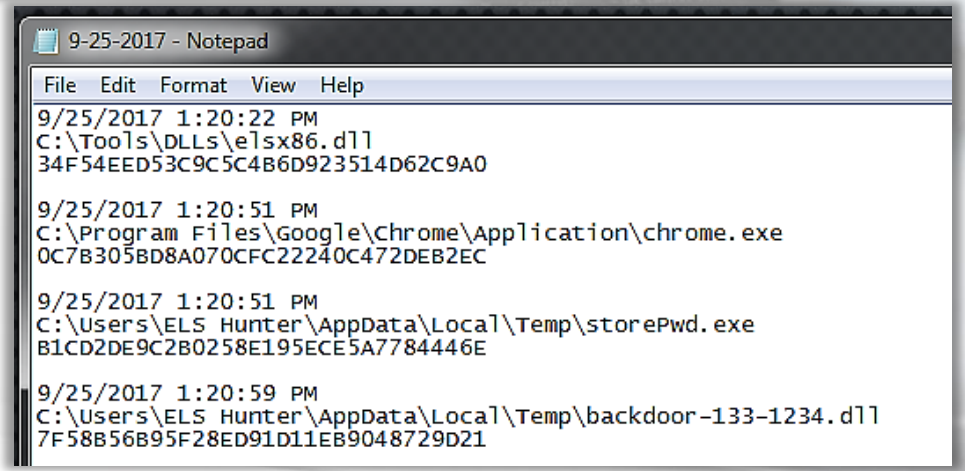
3.2.1 Detection Tools – PE Capture

Based on the indicators shown below, we can already confirm that this is malicious.



3.2.1 Detection Tools – PE Capture





Reviewing the logs is useful to determine what was loaded onto the system earlier that day. You might catch something that you didn't know was loaded since the GUI is displaying information in real time.

3.2.2 Detection Tools – ProcScan.rb

ProcScan, which is written in Ruby, can be used to scan process memory looking for code injection. Unfortunately, it only works for 32-bit systems? applications? and does not support 64-bit systems/applications. You can download the tool here.

To run the tool, type the following command: **ruby ProcScan.rb**

```
C:\Users\elslabs\Desktop\ProcScan>ruby ProcScan.rb
** WARN: Scan all processes? This might take some time. Proceeed (yes/no)?: yes
```


https://github.com/abhisek/RandomCode/tree/master/Malware/Process

3.2.2 Detection Tools – ProcScan.rb

Here is the output of the command if it finds code injection:

```
[+] Scanning: C:\Windows\system32\rundll32.exe
[+] Scanning Thread Id: 2516 EIP: 0x76fe64f4
[+] Scanning Thread Stack Frame (Tid: 2516)
[+] Possibly Injected Code Found at StackFrame[2] ThreadId: 2516 Addr: 0x014b2f2
b
[+] Possibly Injected Code Found at StackFrame[3] ThreadId: 2516 Addr: 0x014b362
2
[+] Possibly Injected Code Found at StackFrame[4] ThreadId: 2516 Addr: 0x014b229
c
[+] Possibly Injected Code Found at StackFrame[5] ThreadId: 2516 Addr: 0x014b1af
0
[+] Possibly Injected Code Found at StackFrame[6] ThreadId: 2516 Addr: 0x014b278
f
[+] Possibly Injected Code Found at StackFrame[7] ThreadId: 2516 Addr: 0x014b280
a
[+] Possibly Injected Code Found at StackFrame[8] ThreadId: 2516 Addr: 0x0153865
f
[+] Possibly Injected Code Found at StackFrame[9] ThreadId: 2516 Addr: 0x015385e
6
[+] Possibly Injected Code Found at StackFrame[10] ThreadId: 2516 Addr: 0x003b00
23
[+] Possibly Injected Code Found at StackFrame[11] ThreadId: 2516 Addr: 0x000e9c
2f
[+] Possibly Injected Code Found at StackFrame[12] ThreadId: 2516 Addr: 0xffffff
ff859710ac
[+] Scanning Thread Id: 1048 EIP: 0x76fe64f4
[+] Scanning Thread Stack Frame (Tid: 1048)
[+] Scanning Thread Id: 3592 EIP: 0x76fe64f4
[+] Scanning Thread Stack Frame (Tid: 3592)
```

3.2.2 Detection Tools – ProcScan.rb

The tool is alerting us that there is possible code injection within thread id 2516 of the rundll32 process.

Unfortunately, the tool doesn't give us the PID within the same output, but this can easily be obtained using PowerShell.

3.2.2 Detection Tools – ProcScan.rb

Simply type **get-process** or **ps** (alias) to retrieve a list of the processes running on the system.

```
PS C:\Users\elslabs\Desktop> ps

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     22       2     1800       2268    31     0.00   3156 cmd
     50       3     1640       7372    50     0.05   1836 conhost
     78       4     1280       8324    71     0.69   2324 conhost
     49       3      880       3824    47     0.00   3992 conhost
     51       3     1680       6616    50     1.70   4080 conhost
    536       5     1292       3988    35     0.48    328 csrss
    308       8    10020       5092    53     4.29    380 csrss
    189       8     2920       6668    35     1.15   1336 dllhost
    144       8    87452     109160   168    27.86   2732 dwm
    862      27    39368      77004   253    18.06   2756 explorer
      0       0        0         24     0           0 Idle
    565      10     2708       6084    32     4.24    496 lsass
    139       4     1152       3004    14     0.05    504 lsm
    143       9     2396       5192    40     0.22   2092 msdtc
     77       4     1536       4544    58     0.03   3368 notepad
    360      11    39660      42288   173     1.06   2788 powershell
    313      10    22736      33540   160     1.17   4072 powershell
    338      16    85400      94864   292     5.19   2384 powershell_ise
    142       6    12960       8964   116     0.05   3844 PresentationFontCache
    240       7     4864       7292    57 ...04.24   2980 rundll32
```

3.2.2 Detection Tools – ProcScan.rb

You can also confirm the thread id of the process using PowerShell.

You can use the following command:

ps | % {$_.Name ; $_.Threads} | % {"`t{0}" –f $_.ID}}

3.2.3 Detection Tools – Meterpreter Payload Detection

The next tool is called **Meterpreter Payload Detection**. As the name of the tool implies, it will scan? all the running processes on the system to detect Meterpreter.

You can download the tool here.

https://github.com/DamonMohammadbagher/Meterpreter_Payload_Detection

3.2.3 Detection Tools – Meterpreter Payload Detection

You run the tool by simply executing the binary from an elevated command prompt.

```
c:\Tools\Meterpreter_Payload_Detection_v1.0.0.5>Meterpreter_Payload_Detection.exe


[#] Meterpreter Payload Detection
[#] IDS-IPS Version: 1.0.0.5
[#] Console version Published by Damon Mohammadbagher
[#] API code and Meterpreter Signature by Rohan Vazarkar, David Bitner
[#] 9/25/2017 2:21:40 PM Started time
[#] IDS Mode only

Scanning 44 process
14:21:40.4548902 : 1600 vmtoolsd  is OK
14:21:40.6108904 : 3260 PECapture  is OK
14:21:40.9072910 : 3732 PresentationFontCache  is OK
14:21:41.1724914 : 1328 svchost  is OK
14:21:41.2816916 : 704 vmacthlp  is OK
14:21:41.5000920 : 524 lsass  is OK
14:21:42.1084931 : 2748 SearchIndexer  is OK
14:21:42.2176933 : 2656 notepad  is OK
14:21:42.3112934 : 2208 cmd  is OK
14:21:42.4516937 : 516 services  is OK
```

3.2.3 Detection Tools – Meterpreter Payload Detection

Similar to **PE Capture**, it's a live capture, so the tool will continually run and alert you of a Meterpreter session in memory, as long as that Meterpreter session is active.

3.2.3 Detection Tools – Meterpreter Payload Detection

Here is the output from the tool if it detects a running Meterpreter session in memory.

```
        Process BaseAddress :  131072

14:32:15.7192060
        Warning : Meterpreter Process Found in Memory !!!
        Infected Process: rundll32 :  3808
        Process EntryPointAddress :  137112

        Infected Memory bytes :
        8C-8B-9B-9E-8F-96-A0-8C-86-8C-A0-8F-8D-90-9C-9A-8C-8C-A0-98-
        9A-8B-8F-96-9B-FF-FF-FF-8C-8B-9B-9E-8F-96-A0-8C-86-8C-A0-8F-
        8D-90-9C-9A-8C-8C-A0-98-9A-8B-A0-96-91-99-90-FF-8C-8B-9B-9E-
        8F-96-A0-8C-86-8C-A0-8F-8D-90-9C-9A-8C-8C-A0-88-9E-96-8B-FF-
        8C-8B-9B-9E-8F-96-A0-8C-86-8C-A0-8F-8D-90-9C-9A-8C-8C-A0-96-

        Process Arguments :
        rundll32.exe
                Process Thread ID: 3812
                   Tid StartAddress: 0
                Process Thread ID: 3620
                   Tid StartAddress: 2007723096
                Process Thread ID: 3604
                   Tid StartAddress: 2007723096
        Infected Process should be killed : rundll32
        Infected Process path : C:\Windows\system32\rundll32.exe
```

3.2.3 Detection Tools – Meterpreter Payload Detection

NOTE: Don’t be confused by the Thread ID displayed in the output. This process is not the same as the one shown in the **PE Capture** snapshots.

3.2.4 Detection Tools – Reflective Injection Detection

This tool was created to detect reflective DLL injections running in memory by looking for a PE header. The program also dumps what it finds concerning the injected process, as well as other unlinked executable pages to the root folder.

You can download the tool here.

https://github.com/papadp/reflective-injection-detection

3.2.4 Detection Tools – Reflective Injection Detection

You run the tool by simply running the binary from an elevated command prompt.

```
c:\Tools\Reflective-Injection-Detection>reflective-injection-detection.exe
no access to memory at 0x00301000 - 0x00340000 in 3336
no access to memory at 0x014d7000 - 0x014f0000 in 3336
no access to memory at 0x07f64000 - 0x07fa0000 in 3336
no access to memory at 0x7ff41000 - 0x7ff50000 in 3336
no access to memory at 0x7ff51000 - 0x7ffa0000 in 3336
PE header found at 0x00870000 in 3808.
PE header found at 0x008a0000 in 3808.
PE header found at 0x01510000 in 3808.
PE header found at 0x01740000 in 3808.
no access to memory at 0x00731000 - 0x00770000 in 3544
16 pages found.
```

3.2.4 Detection Tools – Reflective Injection Detection

You may recall from the output from Meterpreter Payload Detection that the process with a running Meterpreter session is PID 3808.

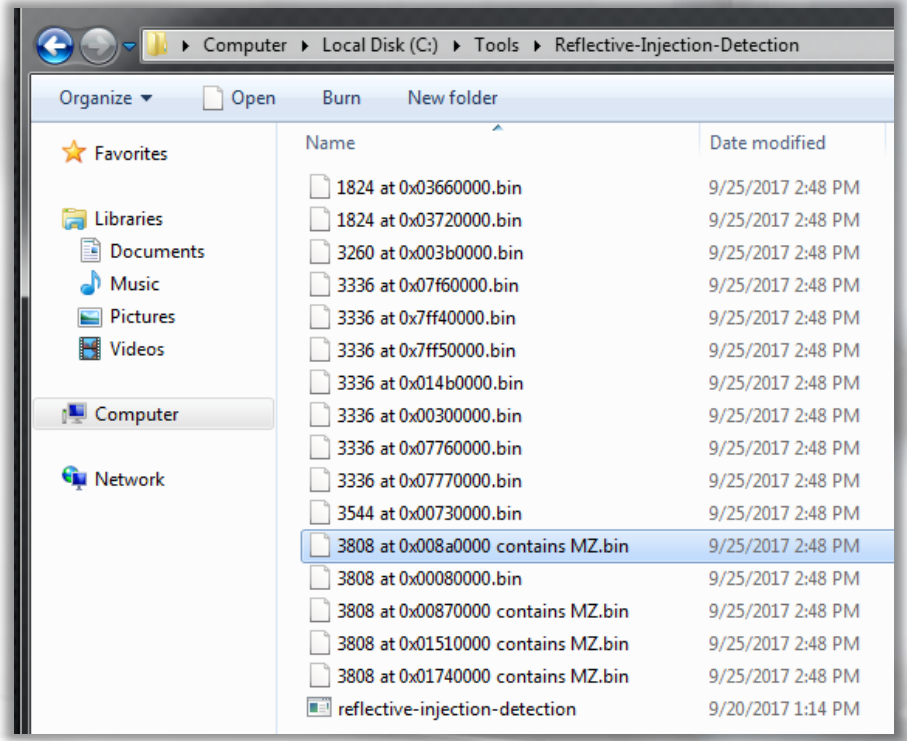
The Reflection Injection Detection tool successfully alerts us about this rundll32 process.

3.2.4 Detection Tools – Reflective Injection Detection

If we navigate to the root folder of the Tool, we will find everything the tool dumped for us, so we can further analyze the artifacts.

Each of the files are named as the PID. This will allow us to easily correlate the file with its process.

You may also notice how the files dumped from process 3808 indicate that 'MZ' was found by listing that information within the name of the dump.



3.2.5 Detection Tools – PowerShell Arsenal

*"**PowerShellArsenal** is a PowerShell module used to aid a reverse engineer. The module can be used to disassemble managed and unmanaged code, perform .NET malware analysis, analyze/scrape memory, parse file formats and memory structures, obtain internal system information, etc."*

https://github.com/mattifestation/PowerShellArsenal

3.2.5 Detection Tools – PowerShell Arsenal

Remember that with the previous tool, **Reflective Injection Detection**, the output gives us the base address and the PID.

To link the output from that tool and the output from PowerShell Arsenal, we will run Reflective Injection Detection again and capture the output. Afterward, we will run the cmdlet **Find-ProcessPEs** from PowerShell Arsenal.

3.2.5 Detection Tools – PowerShell Arsenal

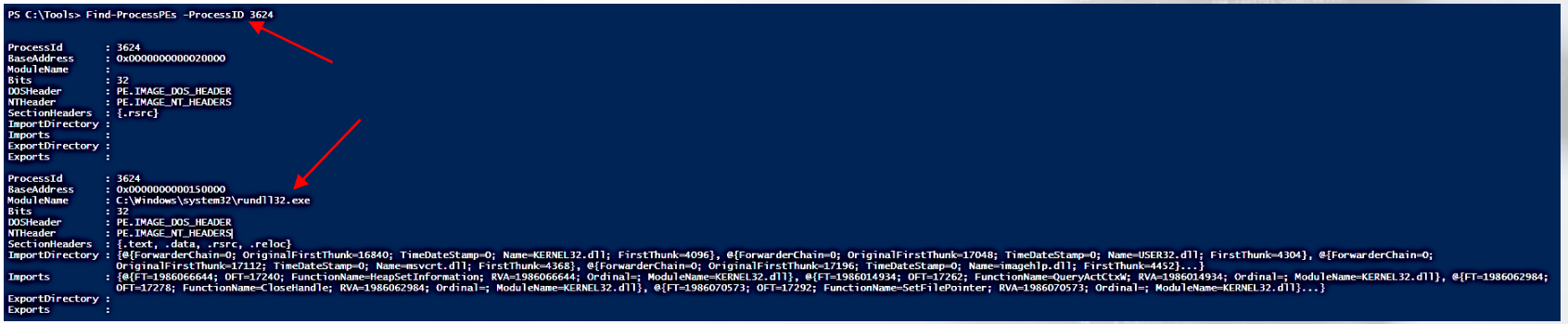
Output from Reflective Injection Detection

```
c:\Tools\Reflective-Injection-Detection>reflective-injection-detection.exe
no access to memory at 0x003a1000 - 0x003e0000 in 2472
no access to memory at 0x005f1000 - 0x00630000 in 2472
no access to memory at 0x04195000 - 0x041b0000 in 2472
no access to memory at 0x7ff41000 - 0x7ff50000 in 2472
no access to memory at 0x7ff51000 - 0x7ff58000 in 2472
no access to memory at 0x7ff59000 - 0x7ffa0000 in 2472
PE header found at 0x004f0000 in 3624.
PE header found at 0x00520000 in 3624.
PE header found at 0x00660000 in 3624.
PE header found at 0x007d0000 in 3624.
15 pages found.
```

The suspicious process is ID? PID? 3624 and we see 4 base addresses displayed in the output. Now, let's run Find-ProcessPEs and compare the output.

3.2.5 Detection Tools – PowerShell Arsenal

In this case, the syntax is: **Find-ProcessPEs –ProcessID 3624**



3.2.5 Detection Tools – PowerShell Arsenal

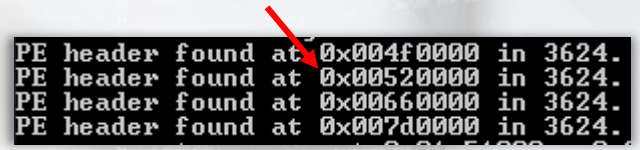
We see that **Find-ProcessPEs** also gives us the same information as far as the base address, but this cmdlet also gives us a bit more.

```
PE header found at 0x004f0000 in 3624.
PE header found at 0x00520000 in 3624.
PE header found at 0x00660000 in 3624.
PE header found at 0x007d0000 in 3624.
```

```
WARNING: Image at address 0x00000000004F0000 was not mapped with LoadLibrary[Ex]. It is possible that malicious code reflectively loaded this module!
WARNING: Failed to read .data section of module 0x00000000004F0000.
WARNING: Failed to read .reloc section of module 0x00000000004F0000.
ProcessId       : 3624
BaseAddress     : 0x00000000004F0000
ModuleName      :
Bits            : 32
DOSHeader       : PE.IMAGE_DOS_HEADER
NTHeader        : PE.IMAGE_NT_HEADERS
SectionHeaders  : {.text, .rdata, .data, .reloc}
ImportDirectory :
Imports         :
ExportDirectory :
Exports         :
```

3.2.5 Detection Tools – PowerShell Arsenal

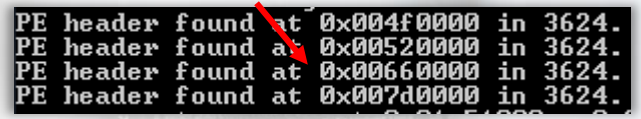
Here we can see detailed output for the next detected PE.




```
WARNING: Image at address 0x0000000000520000 was not mapped with LoadLibrary[Ex]. It is possible that malicious code reflectively loaded this module!
ProcessId       : 3624
BaseAddress     : 0x0000000000520000
ModuleName      : metsrv.dll
Bits            : 32
DOSHeader       : PE.IMAGE_DOS_HEADER
NTHeader        : PE.IMAGE_NT_HEADERS
SectionHeaders  : {.text, .rdata, .data, .reloc}
ImportDirectory : {@{ForwarderChain=0; OriginalFirstThunk=158588; TimeDateStamp=0; Name=WS2_32.dll; FirstThunk=135832}, @{ForwarderChain=0; OriginalFirst
                  OriginalFirstThunk=158548; TimeDateStamp=0; Name=WININET.dll; FirstThunk=135792}, @{ForwarderChain=0; OriginalFirstThunk=158492; TimeDa
Imports         : {@{FT=2009819174; OFT=-2147483632; FunctionName=; RVA=2009819174; Ordinal=16; ModuleName=WS2_32.dll}, @{FT=2009818783; OFT=-2147483630;
                  FunctionName=; RVA=2009804119; Ordinal=14; ModuleName=WS2_32.dll}, @{FT=2009806809; OFT=-2147483536; FunctionName=; RVA=2009806809; Ord
ExportDirectory : @{AddressOfNames=155516; NumberOfNames=81; AddressOfFunctions=155192; Base=1; MinorVersion=0; Name=metsrv.dll; Characteristics=0; Addre
Exports         : {@{ForwardedName=; RVA=31149; Ordinal=81; FunctionName=_scheduler_waitable_thread@4}, @{ForwardedName=; RVA=30960; Ordinal=80; Function
                  FunctionName=scheduler_insert_waitable}, @{ForwardedName=; RVA=30597; Ordinal=78; FunctionName=scheduler_initialize}...}
```

3.2.5 Detection Tools – PowerShell Arsenal

Here we can see detailed output for the next detected PE.




```
WARNING: Image at address 0x0000000000660000 was not mapped with LoadLibrary[Ex]. It is possible that malicious code reflectively loaded this module!
ProcessId       : 3624
BaseAddress     : 0x0000000000660000
ModuleName      : ext_server_priv.x86.dll
Bits            : 32
DOSHeader       : PE.IMAGE_DOS_HEADER
NTHeader        : PE.IMAGE_NT_HEADERS
SectionHeaders  : {.text, .rdata, .data, .reloc}
ImportDirectory : {@{ForwarderChain=0; OriginalFirstThunk=105448; TimeDateStamp=0; Name=PSAPI.DLL; FirstThunk=86508}, @{ForwarderChain=0; OriginalFirst
                  OriginalFirstThunk=104956; TimeDateStamp=0; Name=ADVAPI32.dll; FirstThunk=86016}}
Imports         : {@{FT=1977750948; OFT=105476; FunctionName=GetModuleBaseNameA; RVA=1977750948; Ordinal=; ModuleName=PSAPI.DLL}, @{FT=1977750852; OFT=
                  OFT=105736; FunctionName=WaitForSingleObject; RVA=1986053328; Ordinal=; ModuleName=KERNEL32.dll}, @{FT=1986062984; OFT=105758; Functi
ExportDirectory : @{AddressOfNames=104732; NumberOfNames=5; AddressOfFunctions=104712; Base=1; MinorVersion=0; Name=ext_server_priv.x86.dll; Characteri
Exports         : {@{ForwardedName=; RVA=7829; Ordinal=5; FunctionName=control}, @{ForwardedName=; RVA=4104; Ordinal=4; FunctionName=_ReflectiveLoader@
                  Ordinal=2; FunctionName=GetExtensionName}...}
```

3.2.5 Detection Tools – PowerShell Arsenal

Here we can see detailed output for the next detected PE.


```
WARNING: Image at address 0x00000000007D0000 was not mapped with LoadLibrary[Ex]. It is possible that malicious code reflectively loaded this module!
ProcessId        : 3624
BaseAddress      : 0x00000000007D0000
ModuleName       : ext_server_stdapi.x86.dll
Bits             : 32
DOSHeader        : PE.IMAGE_DOS_HEADER
NTHeader         : PE.IMAGE_NT_HEADERS
SectionHeaders   : {.text, .rdata, .data, .rsrc...}
ImportDirectory  : {@{ForwarderChain=0; OriginalFirstThunk=297496; TimeDateStamp=0; Name=PSAPI.DLL; FirstThunk=258872}, @{ForwarderChain=0; OriginalFirst
                   OriginalFirstThunk=296828; TimeDateStamp=0; Name=IPHLPAPI.DLL; FirstThunk=258204}, @{ForwarderChain=0; OriginalFirstThunk=297512; Time
Imports          : {@{FT=1977750732; OFT=297804; FunctionName=EnumDeviceDrivers; RVA=1977750732; Ordinal=; ModuleName=PSAPI.DLL}, @{FT=1977750804; OFT=29
                   @{FT=1977751134; OFT=297852; FunctionName=GetDeviceDriverFileNameW; RVA=1977751134; Ordinal=; ModuleName=PSAPI.DLL}, @{FT=1930714094;
ExportDirectory  : @{AddressOfNames=296280; NumberOfNames=4; AddressOfFunctions=296264; Base=1; MinorVersion=0; Name=ext_server_stdapi.x86.dll; Character
Exports          : {@{ForwardedName=; RVA=107963; Ordinal=4; FunctionName=_ReflectiveLoader@0}, @{ForwardedName=; RVA=109149; Ordinal=3; FunctionName=Ini
                   @{ForwardedName=; RVA=109177; Ordinal=1; FunctionName=DeinitServerExtension}}
```

3.2.6 Detection Tools – Get-InjectedThread.ps1

This PowerShell tool can aid you on the hunt to detect code injection. This tool will scan active threads on the system. It will retrieve the starting address of certain functions, such as **NTQueryInformationThread**, and if executable code is found, it will flag it as injected.

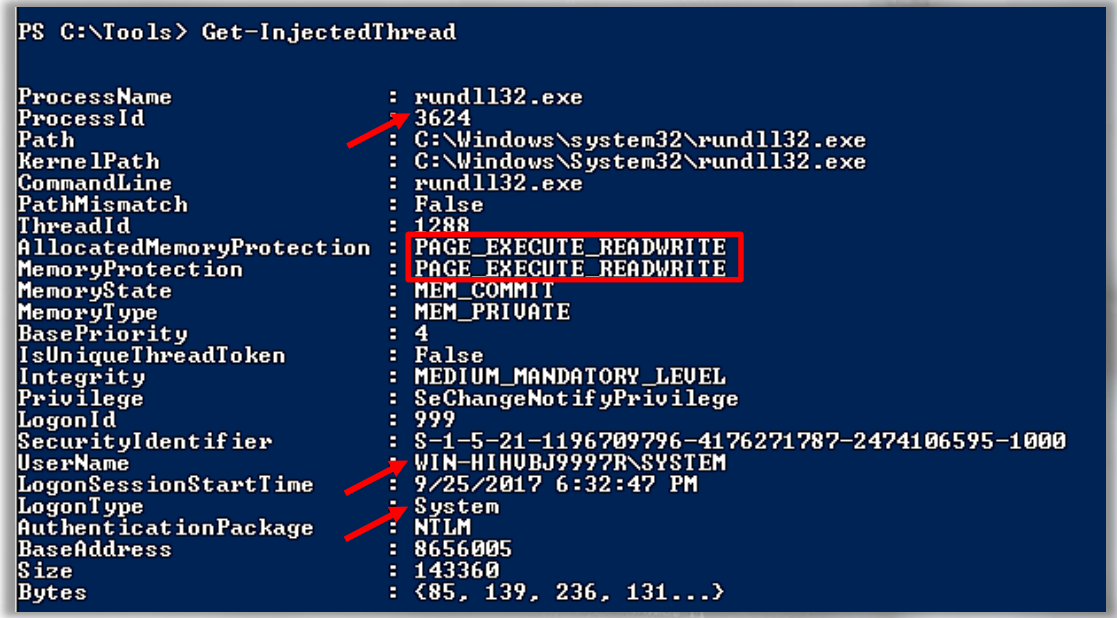
You can download the script here.

https://msdn.microsoft.com/en-us/library/windows/desktop/ms684283(v=vs.85).aspx
https://gist.github.com/jaredcatkinson/23905d34537ce4b5b1818c3e6405c1d2

3.2.6 Detection Tools – Get-InjectedThread.ps1

We will run this tool against the same suspicious process, rundll32 (PID 3624).

We recommend you conduct independent research to fully understand the output from this tool.



Detection Techniques

3.3 Detection Techniques

In this section, we will discuss various techniques to hunt for malware within your network.

Malware authors will try various techniques to ensure that their malware remains undetected. Most of the time, however, the malware in the wild is reused from other malware. This reused malware might be recompiled using a different compiler or modified to remove/add different functionality. In either case, there are techniques to aid us in this hunt.

3.3 Detection Techniques

We will also look at:

- **Fuzzy hashing** and **import hashing** detection techniques to hunt for malware that is reused and is part of an already defined malware family.

- How to detect malware that was already executed on a machine and to correlate various actions that took place on the machine near the time of execution.

3.3.1 Detection Techniques – Fuzzy Hashing

Fuzzy Hashing is a technique where a program, such as SSDeep, computes context triggered piecewise hashes (CTPH). This technique:

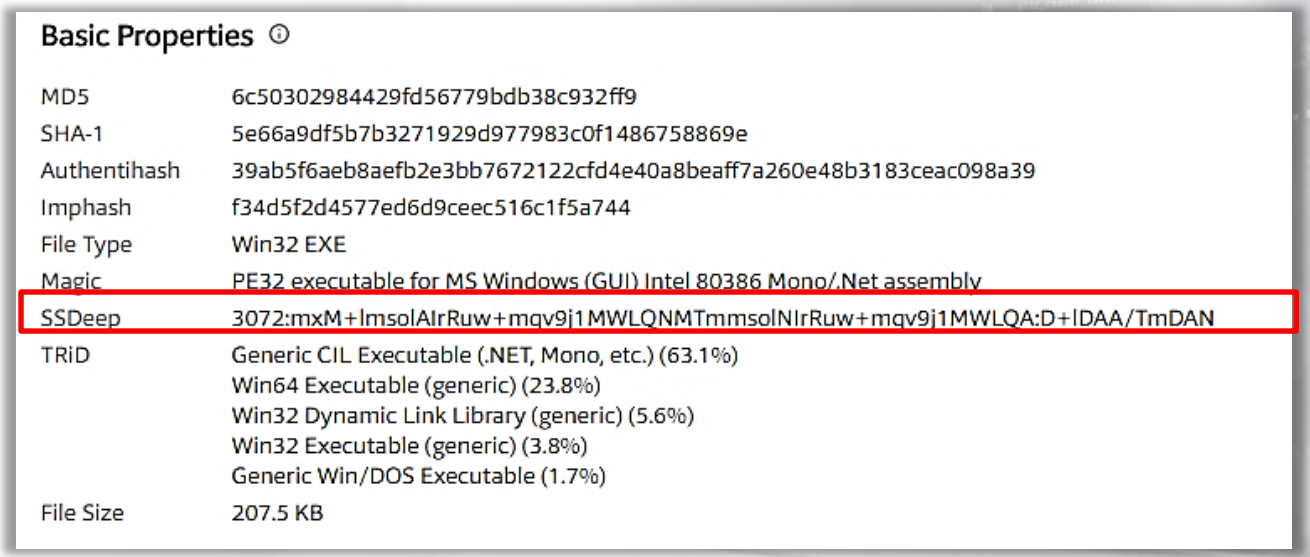
- Can match inputs that have sequences of identical bytes in the same order, although bytes in between the sequences may be different in both content and length.

- Will divide the file into smaller pieces and examine those smaller pieces rather than the file as a whole.

https://ssdeep-project.github.io/ssdeep/index.html

3.3.1 Detection Techniques – Fuzzy Hashing

Virus Total uses **SSDeep**, which performs fuzzy hashing, against files that are uploaded to the platform. The output from **SSDeep** is displayed when the analysis of the uploaded file has completed.



Basic Properties

MD5	6c50302984429fd56779bdb38c932ff9
SHA-1	5e66a9df5b7b3271929d977983c0f1486758869e
Authentihash	39ab5f6aeb8aefb2e3bb7672122cfd4e40a8beaff7a260e48b3183ceac098a39
Imphash	f34d5f2d4577ed6d9ceec516c1f5a744
File Type	Win32 EXE
Magic	PE32 executable for MS Windows (GUI) Intel 80386 Mono/.Net assembly
SSDeep	3072:mxM+lmsolAIrRuw+mqv9j1MWLQNMTmmsolNIrRuw+mqv9j1MWLQA:D+lDAA/TmDAN
TRiD	Generic CIL Executable (.NET, Mono, etc.) (63.1%) Win64 Executable (generic) (23.8%) Win32 Dynamic Link Library (generic) (5.6%) Win32 Executable (generic) (3.8%) Generic Win/DOS Executable (1.7%)
File Size	207.5 KB

3.3.1 Detection Techniques – Fuzzy Hashing

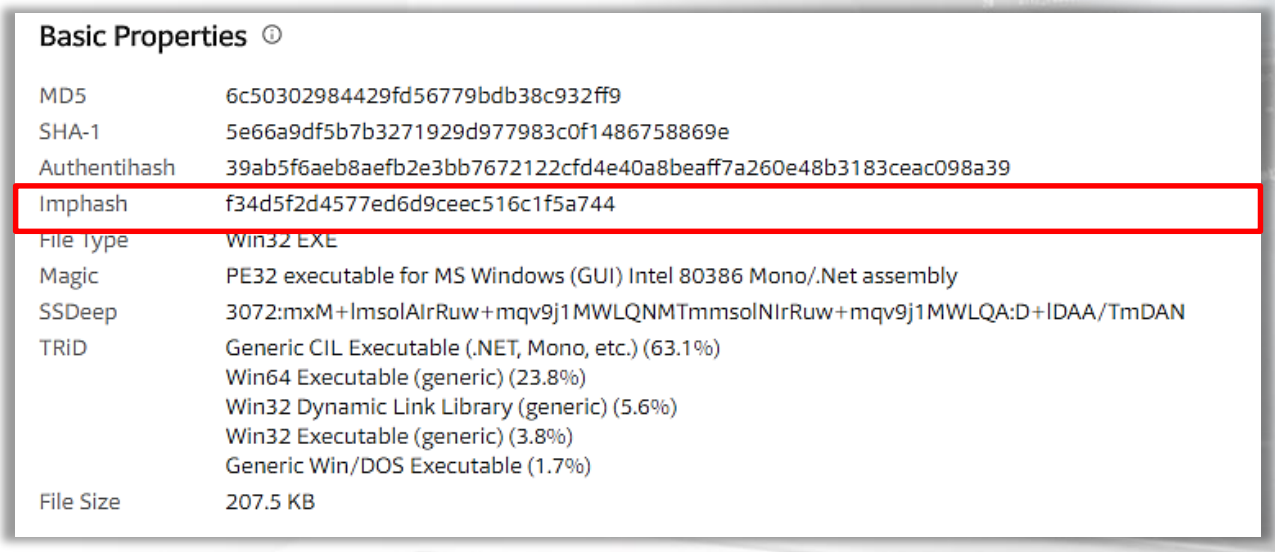
You can read more about this technique in a paper released by the Digital Forensic Research Workshop here. You can also read about an example usage of SSDeep here.

SSDeep is available on GitHub, here.

http://dfrws.org/sites/default/files/session-files/paper-identifying_almost_identical_files_using_context_triggered_piecewise_hashing.pdf
https://dfir.science/2017/07/How-To-Fuzzy-Hashing-with-SSDEEP-(similarity-matching).html
https://github.com/ssdeep-project/ssdeep

3.3.2 Detection Techniques – Import Hashing

The “**imphash**” technique has been coined by Mandiant, and it is yet another technique implemented by Virus Total. It’s part of the output report displayed when a sample has been analyzed, similar to SSDeep.



Basic Properties

MD5	6c50302984429fd56779bdb38c932ff9
SHA-1	5e66a9df5b7b3271929d977983c0f1486758869e
Authentihash	39ab5f6aeb8aefb2e3bb7672122cfd4e40a8beaff7a260e48b3183ceac098a39
Imphash	f34d5f2d4577ed6d9ceec516c1f5a744
File Type	Win32 EXE
Magic	PE32 executable for MS Windows (GUI) Intel 80386 Mono/.Net assembly
SSDeep	3072:mxM+lmsolAIrRuw+mqv9j1MWLQNMTmmsolNIrRuw+mqv9j1MWLQA:D+lDAA/TmDAN
TRiD	Generic CIL Executable (.NET, Mono, etc.) (63.1%) Win64 Executable (generic) (23.8%) Win32 Dynamic Link Library (generic) (5.6%) Win32 Executable (generic) (3.8%) Generic Win/DOS Executable (1.7%)
File Size	207.5 KB

3.3.2 Detection Techniques – Import Hashing

*"One unique way that Mandiant tracks specific threat groups' backdoors is to track portable executable (PE) imports. Imports are the functions that a piece of software (in this case, the backdoor) calls from other files (typically various DLLs that provide functionality to the Windows operating system). To track these imports, Mandiant creates a hash based on library/API names and their specific order within the executable. We refer to this convention as an "**imphash**" (for "**import hash**")."*

http://blog.virustotal.com/2014/02/virustotal-imphash.html

3.3.2 Detection Techniques – Import Hashing

*"Because of the way a PE's import table is generated (and therefore how its **imphash** is calculated), we can use the **imphash** value to identify related malware samples. We can also use it to search for new, similar samples that the same threat group may have created and used."*

You can read more about this technique here and an open source tool to generate PE Import Hashes here.

https://www.mandiant.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html
https://github.com/Neo23x0/ImpHash-Generator

3.3.3 Detection Techniques – Execution Tracing

If you're familiar with forensics, then you know about the ShimCache. The Windows ShimCache was created to track compatibility issues with executed programs and stores various file metadata. You can read more about the ShimCache here.

Five years ago, Mandiant released a tool called ShimCacheParser to gather this metadata within Windows machines to aid them in their investigations. The tool can be downloaded here.

https://www.mandiant.com/blog/threat-research/2015/06/caching_out_the_val.html
https://github.com/mandiant/ShimCacheParser

3.3.3 Detection Techniques – Execution Tracing

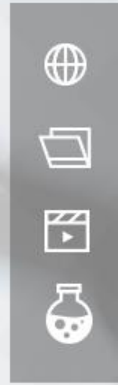
This year, they released an updated tool called AppCompatProcessor, and it contains some analytics to look at the execution trace artifacts obtained from AppCompat / AmCache metadata.

```
AppCompatProcessor (Beta 0.8.0 [2017-03-18T13:25:47])
Feedback, bugs, complaints: matias.bevilacqua@mandiant.com

positional arguments:
  database_file         The database to create or work with
  {load,status,list,dump,search,fsearch,filehitcount,tcorr,ptcorr,tstomp,tstack,stack,leven,reconscan,precon,fevil,hashsearc
h,testset}
    load                Load (or add new) AppCompat / AmCache data
    status              Print status of database
    list                List hosts in database
    dump                Recreate AppCompat/AmCache dump for a given host
    search              Search module
    fsearch             Field search module
    filehitcount        Count # of FileName hits form a user supplied file
    tcorr               Perform temporal execution correlation on a user supplied filename
    ptcorr              Print temporal correlation context for the previously calculated tcorr
    tstomp              Attempt to detect modified last modification timestamps (experimental)
    tstack              Time stacking module (experimental)
    stack               Good old stacking with a sql twist
    leven               Find file name anomalies based on Levenshtein distance
    reconscan           Calculate recon activity in the database
    precon              Print contextual activity to recon commands identified on a host
    fevil               Use temporal correlation on recon sessions to find potential evil (experimental)
    hashsearch          hashsearch module
    testset             Build fake testset database
```

3.3.3 Detection Techniques – Execution Tracing

You can read more about how this tool can be used to detect Temporal Execution Correlation, Time Stacking, etc., here.

You can also download the tool from GitHub here.



https://www.mbeviladev.com/2017/04/appcompatprocessor.html
https://github.com/mbevilacqua/appcompatprocessor

Memory Analysis

3.4 Memory Analysis

Traditional file-system detection techniques are highly unreliable when dealing with memory-resident malware, and therefore it is necessary to perform Memory analysis to detect malware, and also to understand what the purpose and capabilities of the malware are.

3.4 Memory Analysis

Memory forensics can provide unprecedented visibility into the run-time state of a system. It is possible to extract which processes were running, open network connections, and recently executed commands in a manner that is independent of the system. This will reduce the chance of sophisticated malware (rootkits for example) interfering with the results by, for example, modifying them. Moreover, it is likely that critical data exists in memory, such as encryption keys and memory-resident injected code fragments.

3.4 Memory Analysis

Malware often performs injection or system manipulation directly in memory to avoid detection. Some of the techniques often employed, which we'll discuss later are:

- Shellcode injection
- DLL and Reflective DLL injection
- Process hollowing
- API hooking
- Gargoyle

3.4 Memory Analysis

However, before an analyst can analyze the memory, it needs to be acquired (also referred to as dumped, meaning a 1:1 mapping of physical memory to a file, called a memory image) first.

There are two approaches for acquiring memory from a physical device:

- Hardware acquisition
- Software acquisition

3.4 Memory Analysis

Hardware acquisition has the advantage of being more resilient against rootkit modification. It communicates directly the memory controller with no communication to the OS, which you may not be able to trust in the case of a compromised system. Hardware acquisition requires a PCI card to be installed to perform the acquisition.

3.4 Memory Analysis

Software acquisition is used to acquire the object at \\Device \\PhysicalMemory (essentially the Windows memory manager's view of the system). A software tool maps that object and reads its content, which requires kernel mode access to read it. Among some of the requirements for a stable tool are: OS support, memory footprint, ability to capture reserved sections without crashing the system, and output file support.

3.4 Memory Analysis

A drawback of using a software solution is that it will always require process and kernel memory (for itself), as it needs to execute and will therefore overwrite possible evidence. Another drawback is that software solutions are vulnerable to the previously mentioned rootkit modification attacks.

3.4 Memory Analysis

Some of the non-commercial tools available are FTK Imager, DumpIt, and MAGNET RAM Capture.

https://accessdata.com/product-download/ftk-imager-version-4-2-0
https://my.comae.com/
https://www.magnetforensics.com/resources/magnet-ram-capture/

3.4 Memory Analysis

Memory can always be acquired from virtual machines. Some of the VM vendors provide the physical memory file directly if the guest OS has been suspended, or in a snapshot (such as VMware in a .vmem file). Sometimes, additional user interaction is required to generate a memory image, often performed in debugging mode (VirtualBox). Furthermore, memory dumps can be created from a system crash file, hibernation file, and more.

3.4 Memory Analysis

Before jumping into analysis mode, we need to outline what it is that we flag as suspicious on a generic level.

3.4 Memory Analysis

When identifying anomalies in processes, we are interested in:

- Image name - Legitimate process? Spelled correctly?
- Full Path - Appropriate path for system executable? Running from a user or a temp directory?
- Parent process - Is the parent process what you would expect?
- Command line - Do the arguments make sense?
- Start time - Was the process started at boot?
- Security identifier - Do the security identifiers make sense? Why would a system process use a user account SID?

3.4 Memory Analysis

When identifying anomalies in network activity, we are interested in:

- Any process communicating over port 80, 443, or 8080 that is not a web browser
- Any browser not communicating over port 80, 443, or 8080
- Connections to unexplained internal or external IP addresses

3.4 Memory Analysis

When identifying anomalies in network activity, we are interested in (CONTINUED):

- Web requests directly to an IP addresses rather than a domain name
- RDP connections (port 3389), especially if originating from odd IP addresses (e.g. a static IP address assigned to a printer)
- Why does this process have network capability?
- DNS requests for unusual domain names

3.4 Memory Analysis

Moreover, other anomalies are:

- Unlinked processes
- Loaded suspicious DLLs
- Unlinked network connections
- Unmapped memory pages with execute privileges (code injection)
- Hooked API functions
- Known bad heuristics and signatures (e.g. YARA signatures).

3.4 Memory Analysis

Memory analysis is performed through the use of tools specifically designed for that purpose. Within this section, we'll look at these tools that will aid us in it:

- Mandiant's (FireEye) Redline
- Volatility
- Get-InjectedThreat.ps1
- Memdump

https://www.fireeye.com/services/freeware/redline.html
https://github.com/volatilityfoundation/volatility
https://gist.github.com/jaredcatkinson/23905d34537ce4b5b1818c3e6405c1d2
https://github.com/marcosd4h/memhunter

3.4.1 Memory Analysis - Redline

Redline is FireEye's free endpoint security tool that provides host investigative capabilities to find signs of malicious activity through memory and file analysis.

You can download Redline here.

https://www.fireeye.com/services/freeware/redline.html

3.4.1 Memory Analysis - Redline

With Redline, you can:

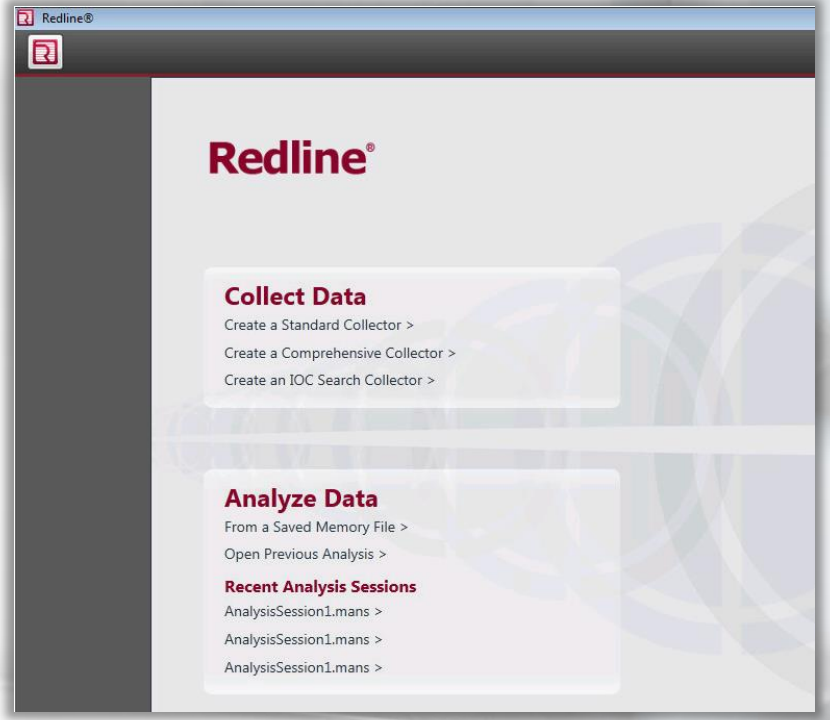
- Thoroughly audit and collect all running processes and drivers from memory, file-system metadata, registry data, event logs, network information, services, tasks and web history.
- Perform Indicators of Compromise (IOC) analysis. Supplied with a set of IOCs, the Redline Portable Agent is automatically configured to gather the data required to perform the IOC analysis and an IOC hit results review.

3.4.1 Memory Analysis - Redline

Redline is a GUI-based tool.

We can create portable agents that can gather a live memory capture of a computer system or many systems. We can also perform an IOC scan against the memory file.

We can also load a memory image and load saved Redline sessions (MANs files).



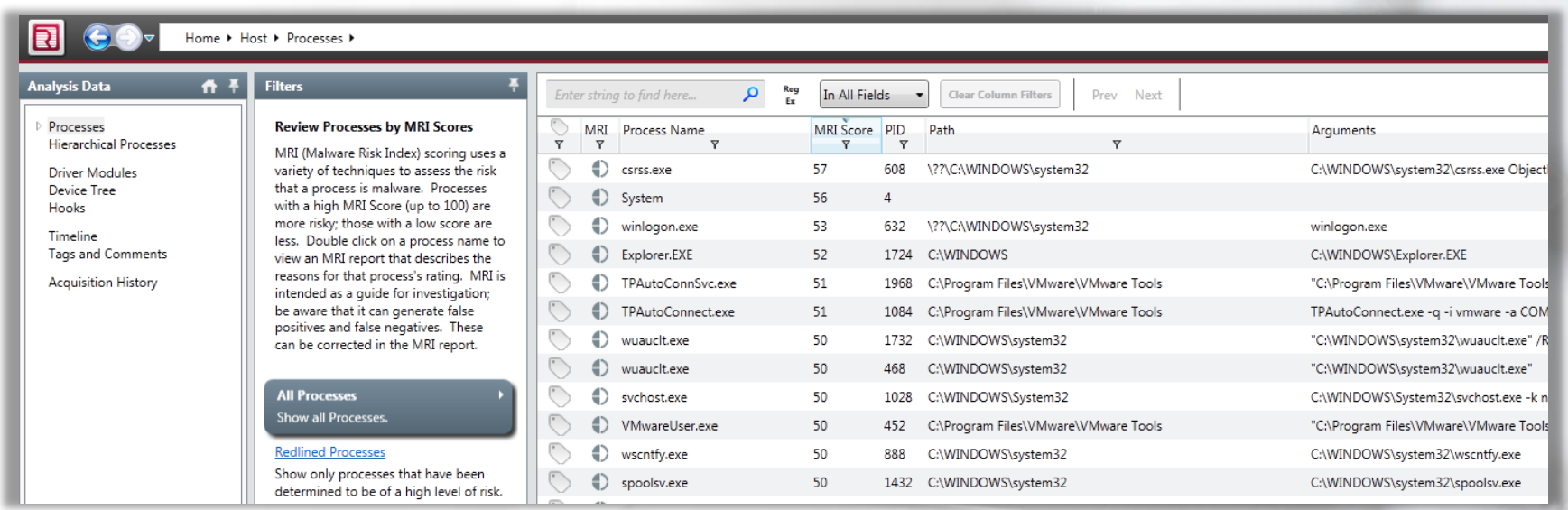
3.4.1 Memory Analysis - Redline

This tool automates the anomaly detection process and gives a quick overview of a particular machine's memory to detect rogue processes, injections, root kits, etc. using the MRI Score Index. Although not always accurate, Redline can still point you in the right direction with your analysis.

The next screen shot will show you the Redline interface along with an explanation as to what MRI Scores are.

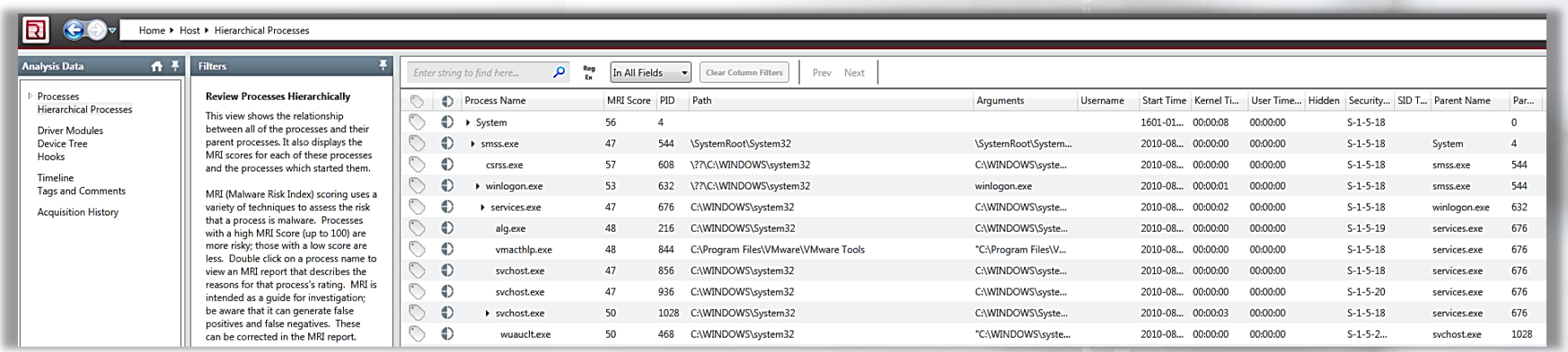
3.4.1 Memory Analysis - Redline

Processes View



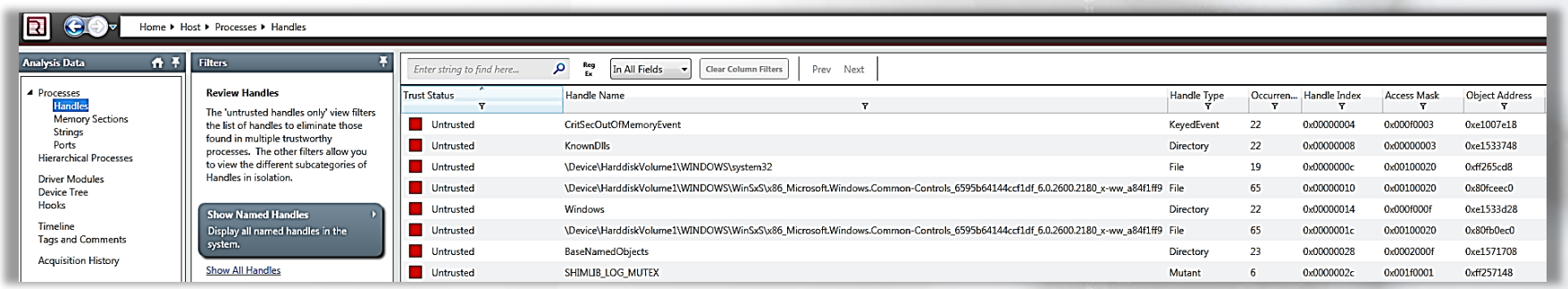
3.4.1 Memory Analysis - Redline

Hierarchical Processes View



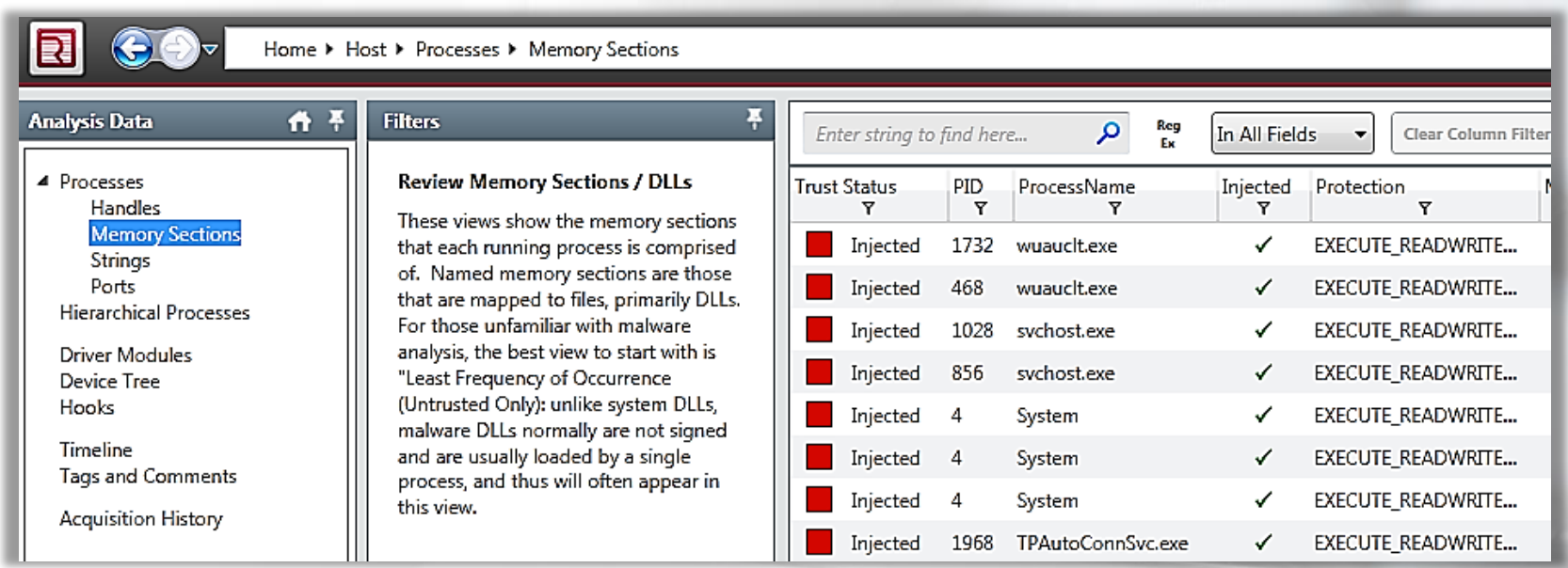
3.4.1 Memory Analysis - Redline

Processes > Handles



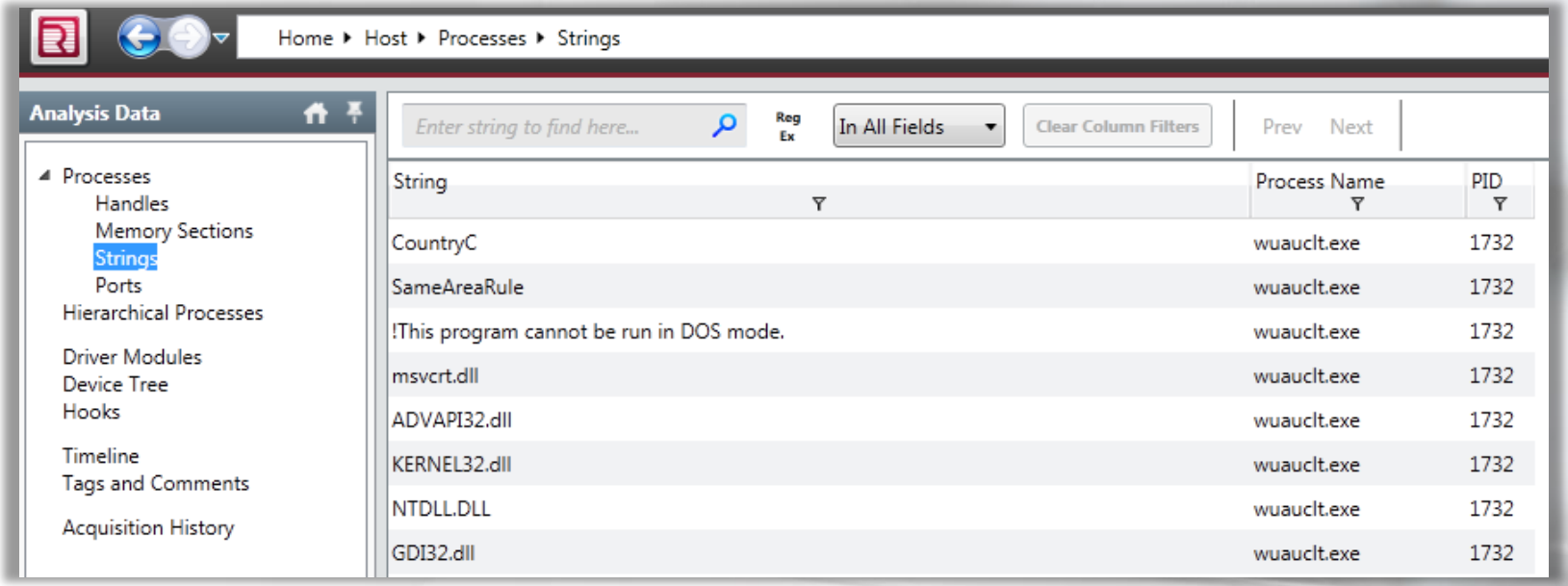
3.4.1 Memory Analysis - Redline

Processes > Memory Sections



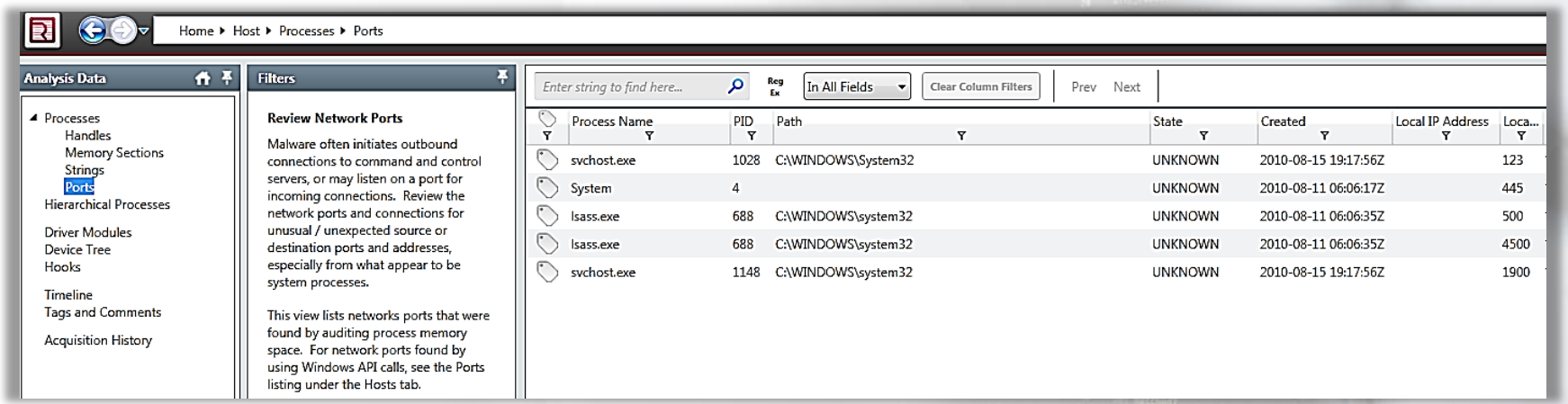
3.4.1 Memory Analysis - Redline

Processes > Strings



3.4.1 Memory Analysis - Redline

Processes > Ports



3.4.1 Memory Analysis - Redline

Again, Redline is good to get a quick look at a machine's memory. This process is known as triaging. When you triage, you're getting a 30,000 foot view of what is going on.

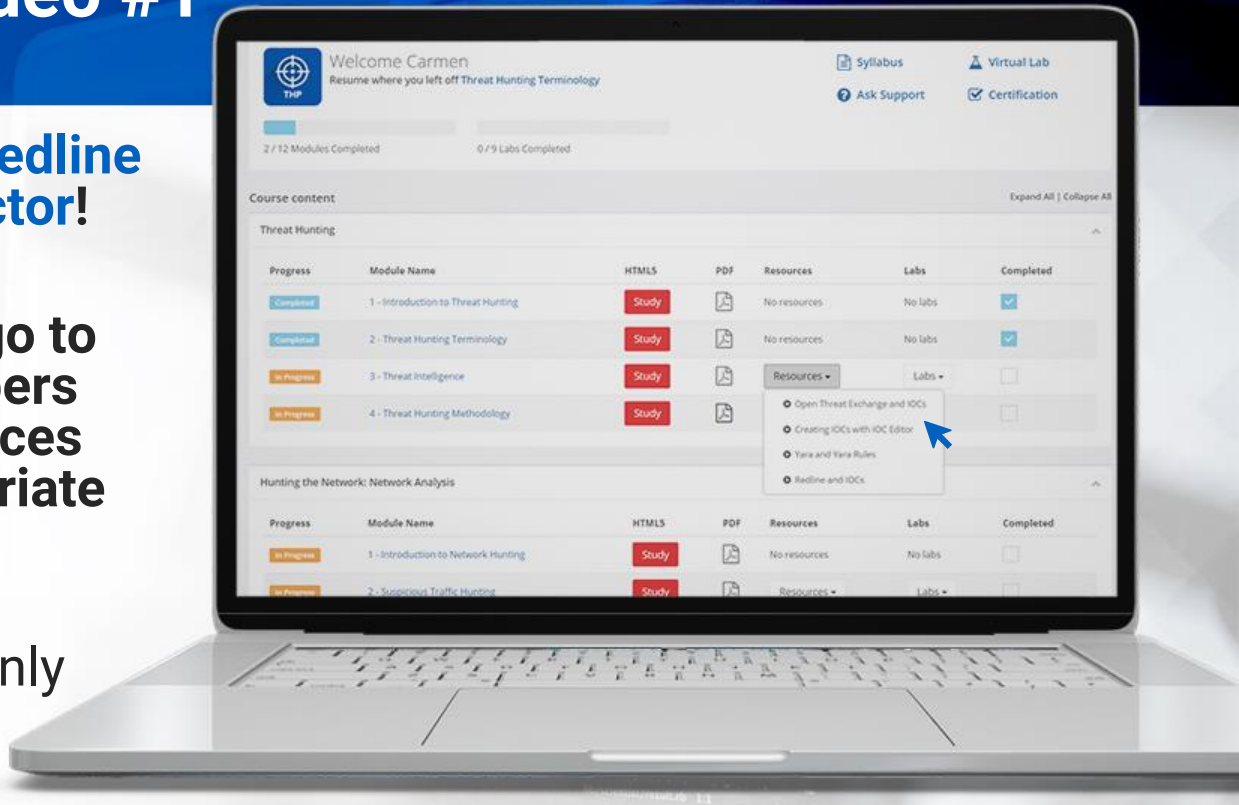
If something is detected as malicious by Redline, then you can take a closer examination with a more advanced tool, such as Volatility.

3.4.1.1 Redline Video #1

Check out the video on **Redline – Create Standard Collector**!

To ACCESS your video, go to the course in your members area and click the resources drop-down in the appropriate module line.

Note that all videos are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.

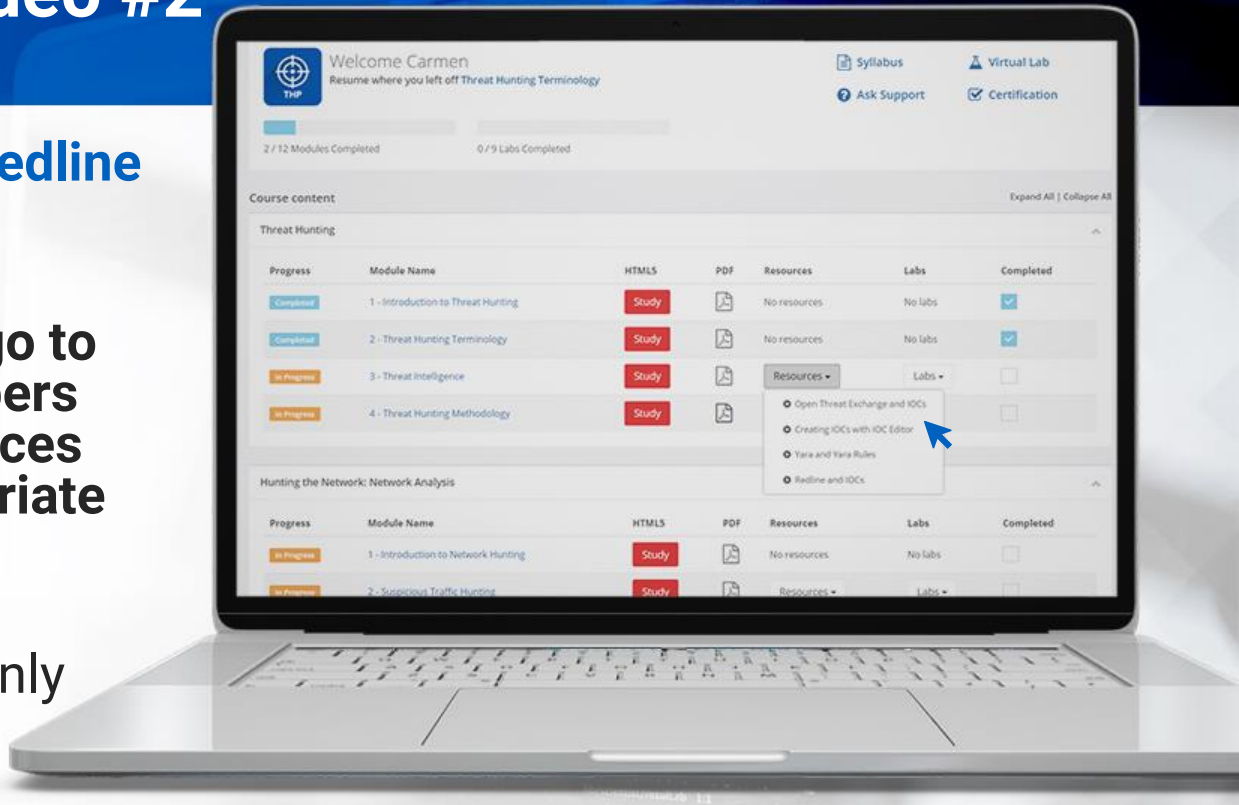


3.4.1.2 Redline Video #2

Check out the video on **Redline – Basic Usage**!

To ACCESS your video, go to the course in your members area and click the resources drop-down in the appropriate module line.

Note that all videos are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.

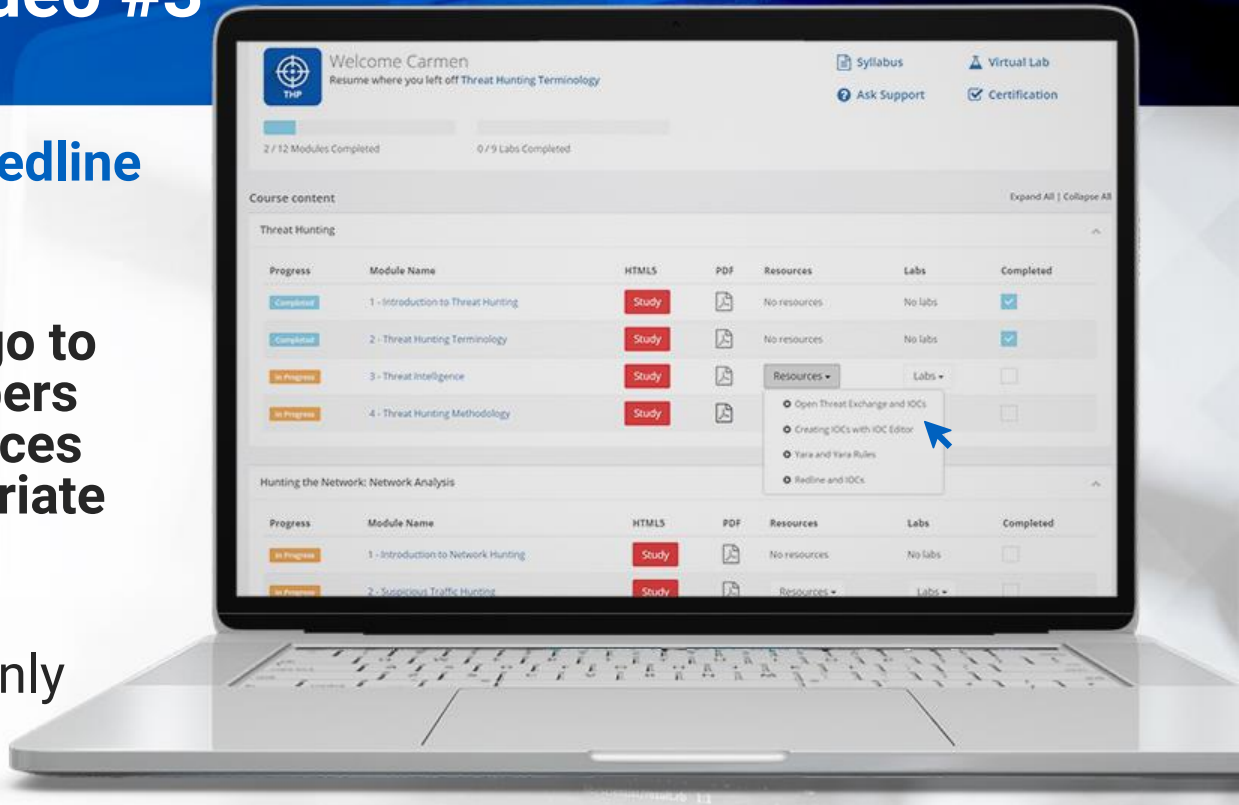


3.4.1.3 Redline Video #3

Check out the video on **Redline – Create Analysis File**!

To ACCESS your video, go to the course in your members area and click the resources drop-down in the appropriate module line.

Note that all videos are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.

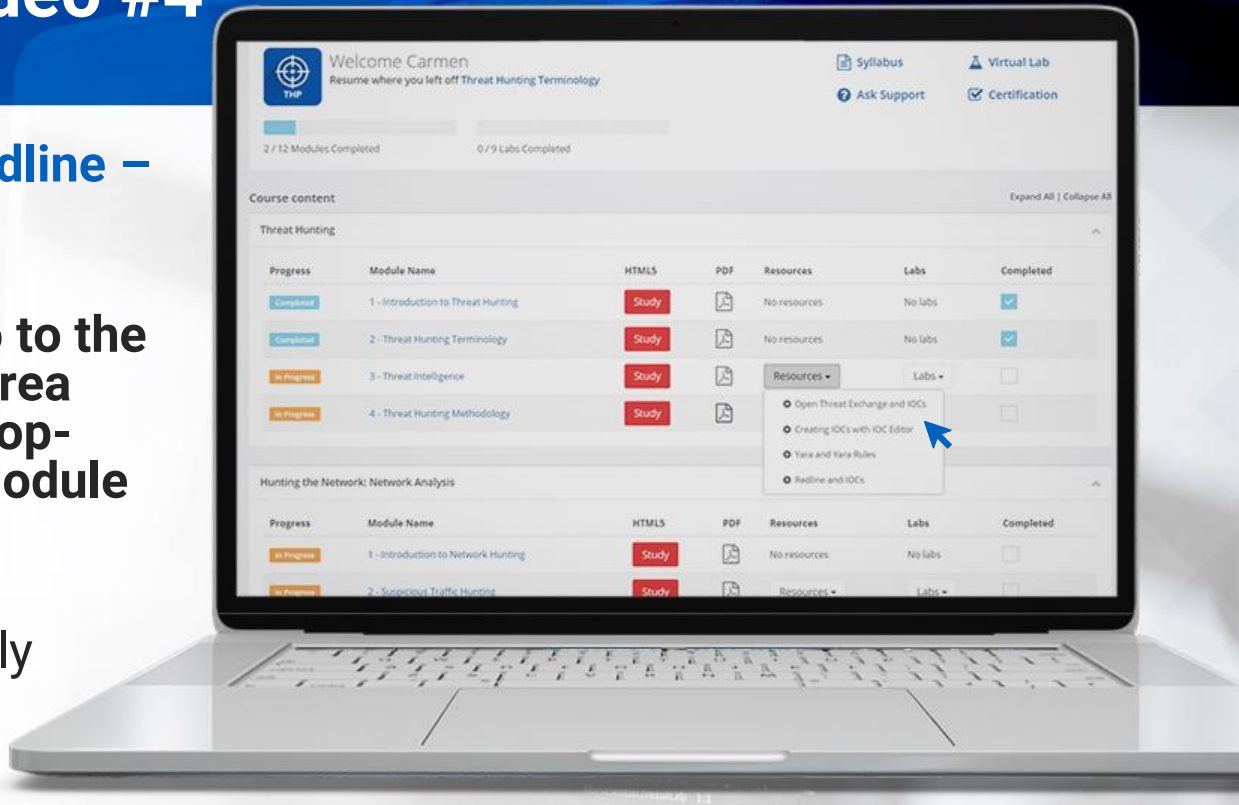


3.4.1.4 Redline Video #4

Check out the video on **Redline – Detecting Code Injection**!

To ACCESS your video, go to the course in your members area and click the resources drop-down in the appropriate module line.

Note that all videos are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.



3.4.2 Memory Analysis - Volatility

"The Volatility Framework is a completely open collection of tools, implemented in Python under the GNU General Public License, for the extraction of digital artifacts from volatile memory (RAM) samples. The extraction techniques are performed completely independent of the system being investigated but offer visibility into the runtime state of the system." – Volatility on GitHub

https://github.com/volatilityfoundation/volatility

3.4.2 Memory Analysis - Volatility

Volatility is not as user friendly as Redline, but is definitely an excellent tool that is worth learning and getting comfortable with. Volatility will be able to detect malicious activity that Redline might miss.

3.4.2 Memory Analysis - Volatility

Volatility is available for Windows, Linux, and Mac OS and is written purely in Python. In order to perform an analysis in Volatility, we need to specify three parameters:

- Memory dump file
- OS Profile
- Plugin (also called Module)

3.4.2 Memory Analysis - Volatility

An Operating System profile is required because each version of an Operating System has different definition and implementation of memory objects, so this tells Volatility how to treat the memory image in order to find data structures in it.

By default, Volatility comes with all existing Windows profiles from Windows XP to Windows 10.

3.4.2 Memory Analysis - Volatility

The plugin is the payload of the command. It tells Volatility what we are looking for in the memory image.

Currently, Volatility supports over 200 plugins by default, and the analyst has the opportunity to extend Volatility's capabilities by developing custom plugins. Some of the plugins are show on the next slide.

3.4.2 Memory Analysis - Volatility

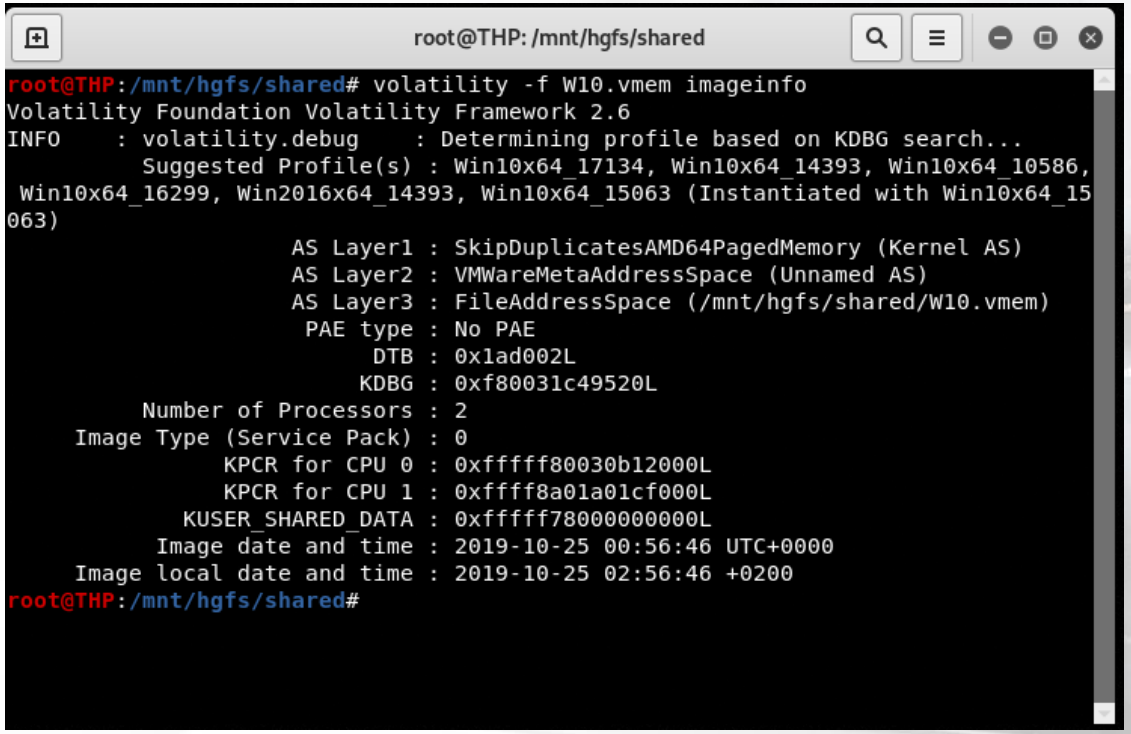
```
Supported Plugin Commands:

        amcache         Print AmCache information
        apihooks        Detect API hooks in process and kernel memory
        atoms           Print session and window station atom tables
        atomscan        Pool scanner for atom tables
        auditpol        Prints out the Audit Policies from HKLM\SECURITY\Policy\PolAdtEv
        bigpools        Dump the big page pools using BigPagePoolScanner
        bioskbd         Reads the keyboard buffer from Real Mode memory
        cachedump       Dumps cached domain hashes from memory
        callbacks       Print system-wide notification routines
        clipboard       Extract the contents of the windows clipboard
        cmdline         Display process command-line arguments
        cmdscan         Extract command history by scanning for _COMMAND_HISTORY
        connections     Print list of open connections [Windows XP and 2003 Only]
        connscan        Pool scanner for tcp connections
        consoles        Extract command history by scanning for _CONSOLE_INFORMATION
        crashinfo       Dump crash-dump information
        deskscan        Poolscaner for tagDESKTOP (desktops)
        devicetree      Show device tree
        dlldump         Dump DLLs from a process address space
        dlllist         Print list of loaded dlls for each process
        driverirp       Driver IRP hook detection
        drivermodule    Associate driver objects to kernel modules
        driverscan      Pool scanner for driver objects
        dumpcerts       Dump RSA private and public SSL keys
        dumpfiles       Extract memory mapped and cached files
        dumpregistry    Dumps registry files out to disk
        editbox         Displays information about Edit controls. (Listbox experimental.)
        envars          Display process environment variables
        eventhooks      Print details on windows event hooks
```

3.4.2 Memory Analysis - Volatility

As mentioned, before starting the analysis, Volatility requires the OS version to be specified as a command line argument.

Often times, as an analyst, you would know that, but in the cases you don't, a helpful plugin is "imageinfo", which identifies (to its best capabilities) the OS version from the memory dump itself as shown on the next slide.

3.4.2 Memory Analysis - Volatility



3.4.2 Memory Analysis - Volatility

When the plugin finishes executing, Volatility presents us back with a list of potential OS profiles, sorted by the most likely one.

In this example, the profile is "Win10x64_17134". Armed with the profile, we can continue and begin the analysis.

3.4.2 Memory Analysis - Volatility

One of the basic functions of Volatility is to list processes running on the system with the plugin "pslist”. In order to locate processes with "pslist", Volatility is locating the doubly-linked list that keeps track of the processes in memory, and displays them back to the user. This is the equivalent of the processes list in task manager on a running Windows system

3.4.2 Memory Analysis - Volatility

Note that the output may include information on processes that have already terminated, which includes their exit time. This can be particularly useful in cases where a process, such as cmd.exe, is used to start a malicious executable and exits afterwards.

An example of this plugin is shown on the next slide.

3.4.2 Memory Analysis - Volatility

```
root@THP:/mnt/hgfs/shared# volatility -f W10.vmem --profile Win10x64_17134 pslist
Volatility Foundation Volatility Framework 2.6
Offset(V)          Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                          Exit
------------------ -------------------- ------ ------ ------ -------- ------ ------ ------------------------------ ------------------------------
0xffffe48904ac5040 System                    4      0     96        0 ------      0 2019-10-24 04:13:11 UTC+0000
0xffffe48904b51040 Registry                 88      4      3        0 ------      0 2019-10-24 04:13:02 UTC+0000
0xffffe48905e92040 smss.exe                280      4      2        0 ------      0 2019-10-24 04:13:11 UTC+0000
0xffffe48905e34580 csrss.exe               384    376     10        0      0      0 2019-10-24 04:13:12 UTC+0000
0xffffe489067cc080 wininit.exe             468    376      1        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe489063e7080 csrss.exe               476    460     12        0      1      0 2019-10-24 04:13:13 UTC+0000
0xffffe489067ef080 winlogon.exe            536    460      6        0      1      0 2019-10-24 04:13:13 UTC+0000
0xffffe48905293080 services.exe            604    468      9        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe48905384080 lsass.exe               620    468      8        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe48906886580 svchost.exe             724    604      2        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe48906882580 fontdrvhost.ex          748    468      5        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe48906884580 fontdrvhost.ex          756    536      5        0      1      0 2019-10-24 04:13:13 UTC+0000
0xffffe4890684b080 svchost.exe             764    604     18        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe48906870580 svchost.exe             864    604     20        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe48906848580 svchost.exe             908    604      6        0      0      0 2019-10-24 04:13:13 UTC+0000
0xffffe48906ac8080 dwm.exe                 968    536     12        0      1      0 2019-10-24 04:13:14 UTC+0000
0xffffe48906ac6400 LogonUI.exe             976    536      0 --------      1      0 2019-10-24 04:13:14 UTC+0000   2019-10-25 00:46:45 UTC+0000
0xffffe48906b77580 svchost.exe             600    604     16        0      0      0 2019-10-24 04:13:14 UTC+0000
0xffffe48906b75580 svchost.exe             796    604      4        0      0      0 2019-10-24 04:13:14 UTC+0000
```

3.4.2 Memory Analysis - Volatility

Malware, specifically rootkits, often tries to hide its existence by unlinking itself from this list (amongst other techniques), in which case the process will not be shown in the output by pslist.

Fortunately, in memory, we can locate processes by other means, such as searching through the memory dump and finding data structures that match that of an "_EPROCESS", the representative structure of a process in memory. By doing so, we can identify even hidden processes. For this purpose, we have at our disposal the plugin "psscan".

3.4.2 Memory Analysis - Volatility

"Psscan" scans the entire memory dump and reports on any identified objects that have the structure of an _EPROCESS.

In some cases, this plugin may return false positives, and also processes that have finished execution some time ago (in some occasions, even from a previous reboot).

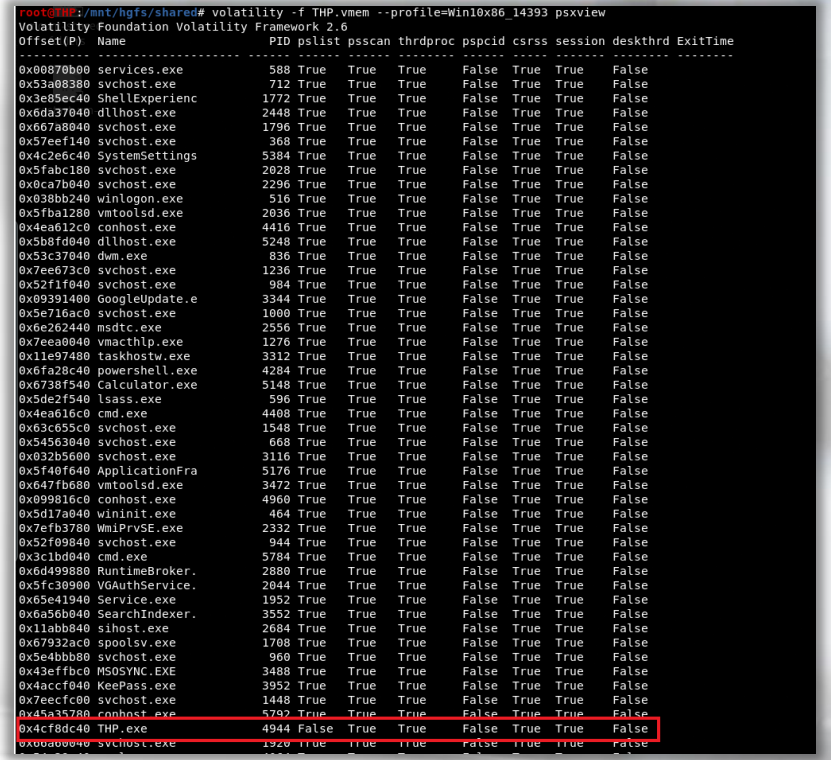
3.4.2 Memory Analysis - Volatility

Yet another, and even more powerful plugin that we may utilize to identify hidden processes, is "psxview", which uses multiple techniques for finding processes in memory. It then reports the output in a single view by displaying whether or not a certain process exists for each of the detection techniques.

The next slide shows the output of "psxview“.

3.4.2 Memory Analysis - Volatility

Note that “THP.exe” was hidden from “pslist” among others.



```
root@THP:/mnt/hgfs/shared# volatility -f THP.vmem --profile=Win10x86_14393 psxview
Volatility Foundation Volatility Framework 2.6
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd ExitTime
---------- -------------------- ------ ------ ------ -------- ------ ----- ------- -------- --------
0x00870b00 services.exe            588 True   True   True     False  True  True    False
0x53a08380 svchost.exe             712 True   True   True     False  True  True    False
0x3e85ec40 ShellExperienc         1772 True   True   True     False  True  True    False
0x6da37040 dllhost.exe            2448 True   True   True     False  True  True    False
0x667a8040 svchost.exe            1796 True   True   True     False  True  True    False
0x57eef140 svchost.exe             368 True   True   True     False  True  True    False
0x4c2e6c40 SystemSettings         5384 True   True   True     False  True  True    False
0x5fabc180 svchost.exe            2028 True   True   True     False  True  True    False
0x0ca7b040 svchost.exe            2296 True   True   True     False  True  True    False
0x038bb240 winlogon.exe            516 True   True   True     False  True  True    False
0x5fba1280 vmtoolsd.exe           2036 True   True   True     False  True  True    False
0x4ea612c0 conhost.exe            4416 True   True   True     False  True  True    False
0x5b8fd040 dllhost.exe            5248 True   True   True     False  True  True    False
0x53c37040 dwm.exe                 836 True   True   True     False  True  True    False
0x7ee673c0 svchost.exe            1236 True   True   True     False  True  True    False
0x52f1f040 svchost.exe             984 True   True   True     False  True  True    False
0x09391400 GoogleUpdate.e         3344 True   True   True     False  True  True    False
0x5e716ac0 svchost.exe            1000 True   True   True     False  True  True    False
0x6e262440 msdtc.exe              2556 True   True   True     False  True  True    False
0x7eea0040 vmacthlp.exe           1276 True   True   True     False  True  True    False
0x11e97480 taskhostw.exe          3312 True   True   True     False  True  True    False
0x6fa28c40 powershell.exe         4284 True   True   True     False  True  True    False
0x6738f540 Calculator.exe         5148 True   True   True     False  True  True    False
0x5de2f540 lsass.exe               596 True   True   True     False  True  True    False
0x4ea616c0 cmd.exe                4408 True   True   True     False  True  True    False
0x63c655c0 svchost.exe            1548 True   True   True     False  True  True    False
0x54563040 svchost.exe             668 True   True   True     False  True  True    False
0x032b5600 svchost.exe            3116 True   True   True     False  True  True    False
0x5f40f640 ApplicationFra         5176 True   True   True     False  True  True    False
0x647fb680 vmtoolsd.exe           3472 True   True   True     False  True  True    False
0x099816c0 conhost.exe            4960 True   True   True     False  True  True    False
0x5d17a040 wininit.exe             464 True   True   True     False  True  True    False
0x7efb3780 WmiPrvSE.exe           2332 True   True   True     False  True  True    False
0x52f09840 svchost.exe             944 True   True   True     False  True  True    False
0x3c1bd040 cmd.exe                5784 True   True   True     False  True  True    False
0x6d499880 RuntimeBroker.         2880 True   True   True     False  True  True    False
0x5fc30900 VGAuthService.         2044 True   True   True     False  True  True    False
0x65e41940 Service.exe            1952 True   True   True     False  True  True    False
0x6a56b040 SearchIndexer.         3552 True   True   True     False  True  True    False
0x11abb840 sihost.exe             2684 True   True   True     False  True  True    False
0x67932ac0 spoolsv.exe            1708 True   True   True     False  True  True    False
0x5e4bbb80 svchost.exe             960 True   True   True     False  True  True    False
0x43effbc0 MSOSYNC.EXE            3488 True   True   True     False  True  True    False
0x4accf040 KeePass.exe            3952 True   True   True     False  True  True    False
0x7eecfc00 svchost.exe            1448 True   True   True     False  True  True    False
0x45a35780 conhost.exe            5792 True   True   True     False  True  True    False
0x4cf8dc40 THP.exe                4944 False  False  True     False  True  True    False
0x60da0040 svchost.exe            1920 True   True   True     False  True  True    False
```

3.4.2 Memory Analysis - Volatility

In the hunt for malicious processes, often times we attempt to identify anomalies, such as whether a process has been started by an expected Parent Process. For this purpose, we can utilize "pstree" in Volatility, whose output is a dot-aligned listing as shown on the next slide. With this view, we can identify obvious anomalies, such as if the parent process of svchost.exe is not services.exe. Another example would be if notepad.exe is starting PowerShell.

Note that this plugin will not include hidden processes in its output!

3.4.2 Memory Analysis - Volatility

Output of “pstree”:

```
root@THP:/mnt/hgfs/shared# volatility -f THP.vmem --profile=Win10x86_14393 pstree
Volatility Foundation Volatility Framework 2.6
Name                                                  Pid   PPid   Thds   Hnds Time
-------------------------------------------------- ------ ------ ------ ------ ----
 0x9308c040:wininit.exe                               464    352      2      0 2019-10-26 17:08:35 UTC+0000
. 0x8b787b00:services.exe                             588    464      5      0 2019-10-26 17:08:36 UTC+0000
.. 0x9536d040:svchost.exe                            1920    588     12      0 2019-10-26 17:08:45 UTC+0000
.. 0x9532b040:svchost.exe                            1032    588     15      0 2019-10-26 17:08:40 UTC+0000
.. 0xa6179600:svchost.exe                            3116    588      7      0 2019-10-26 17:10:26 UTC+0000
.. 0x9526b5c0:svchost.exe                            1548    588      9      0 2019-10-26 17:08:44 UTC+0000
.. 0xa6029040:dllhost.exe                            2448    588     11      0 2019-10-26 17:08:50 UTC+0000
.. 0x93124040:svchost.exe                             668    588     23      0 2019-10-26 17:08:37 UTC+0000
... 0x8ff0f880:RuntimeBroker.                        2880    668     34      0 2019-10-26 17:20:55 UTC+0000
... 0x8b102c40:SystemSettings                        5384    668     19      0 2019-10-26 17:30:18 UTC+0000
... 0x9f6cec40:SearchUI.exe                          2520    668     44      0 2019-10-26 17:20:56 UTC+0000
```

3.4.2 Memory Analysis - Volatility

Volatility's "netscan" plugin traverses memory and identifies all memory structures that represent a network connection. Similar to "psscan", you may find false positives in its output. However, it may also display connections which are no longer active that are still preserved in memory.

The next thing that we'll look at is code injection. At this point, we assume that the reader is familiar with the basic structure of a Portable Executable (PE) file. If not, you can refer to this link and read more about it.

https://resources.infosecinstitute.com/2-malware-researchers-handbook-demystifying-pe-file/#gref

3.4.2 Memory Analysis - Volatility

In general, executable code resides in the ".text" section of a PE file, both when it’s located on disk and also when loaded into memory. With a few exceptions, this is where executable code should reside.

An Injected code will not show in the text section, as it will be placed on the "heap" of a process. Let's look at the concept of DLL Injection.

3.4.2 Memory Analysis - Volatility

DLL Injection is the process of inserting code into a running process. The code inserted is in the form of a Dynamic Link Library (DLL), mainly because DLLs are meant to be loaded as needed at run time. Although, this does not mean that injection of other types of assembly is not possible, such as executables or simply handwritten shellcode.

3.4.2 Memory Analysis - Volatility

Injecting into SYSTEM process or process from another context (eg. process of another user) requires certain privileges (more specifically, SeDebugPrivilege, which is required to debug and adjust the memory of another process).

This is usually achieved through administrative rights on the machine.

3.4.2 Memory Analysis - Volatility

Although there are multiple varieties of code injection techniques, the most generic one is a 4-step process where the Win32 API is used to provide the necessary functionality. The steps are:

1. Attach to the victim process
2. Allocate memory within the victim process
3. Copy the DLL or the DLL Path into the allocated memory
4. Instruct the process to Execute the DLL

3.4.2 Memory Analysis - Volatility

Each of these steps is associated with one or more Win32 API function calls.

During step 3, the malware author has the option to either inject the path of the DLL on disk and load then execute it, or inject the DLL itself, if the allocated memory in the victim process is large enough.

3.4.2 Memory Analysis - Volatility

A drawback in the steps performed during the injection for malware authors, is that the DLL that is being injected may need to be located on disk and could potentially be caught by Antivirus software. Through static analysis, it may be possible to identify injection capability by just observing the import headers of a PE file (if it is not obfuscated in some way). Unlike a Simple DLL Injection, the power of Reflective DLL Injection comes in that it is able to inject to and execute directly from memory.

3.4.2 Memory Analysis - Volatility

Reflective Injection is a special technique of code injection, where code is injected and loaded from memory, directly in the process itself. This type of injection is often used to further expand the capabilities of a functionality limited stager, by delivering additional modules only when needed. The library loading itself is not registered in any way with the host system, and as a result, it is largely undetectable at both a system and process level.

3.4.2 Memory Analysis - Volatility

Techniques of detecting code injections have been around for a while.

One of the most famous is scanning through private memory regions (the heap of a process) and identifying those that have the executable bit set (RWE or RX), and/or have no memory mapped file present on disk (unmapped binary file).

3.4.2 Memory Analysis - Volatility

An unmapped process binary is an indication of process hollowing.

A detailed explanation and research on process hollowing is available here.

https://cysinfo.com/detecting-deceptive-hollowing-techniques/

3.4.2 Memory Analysis - Volatility

The detective techniques mentioned so far are employed by Volatility's malfind plugin for detecting code injection.

Among other details, in malfind's output we can see:

- Process name and PID where injection was detected
- Offset address of where the injection was detected
- Hex, ASCII, and Disassembly view of the injected area

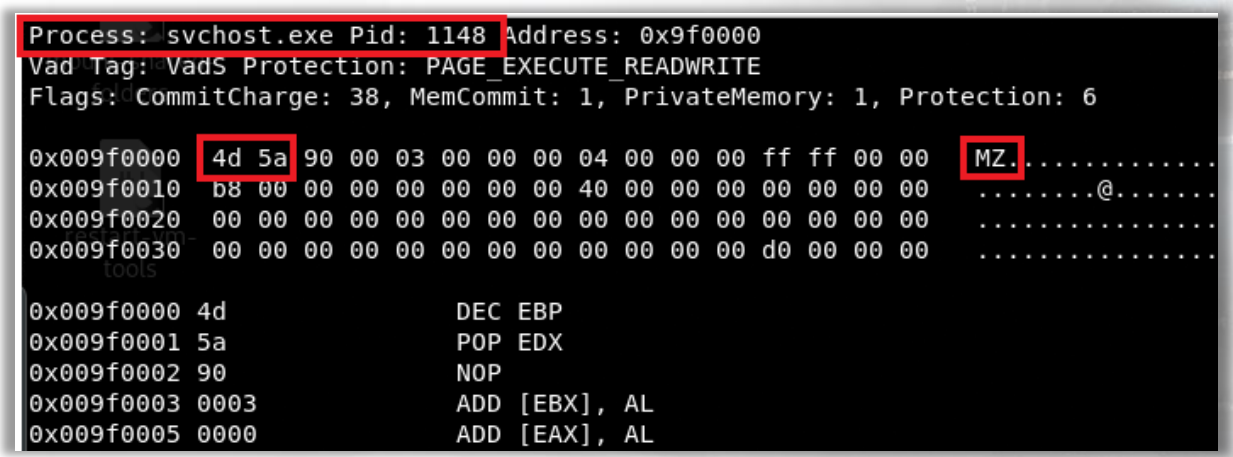
3.4.2 Memory Analysis - Volatility

Injected areas that begin with the "MZ" file header are especially interesting to us – denoting a Windows executable file (which is the case on the picture on the next slide). Of course, the injected area may contain shellcode, which lacks the “MZ” header, and which requires that the analyst further investigate to understand its behavior and purpose.

Note: malfind's output may contain false positives.

3.4.2 Memory Analysis - Volatility

Partial output of “malfind” – “MZ” header detected.



3.4.2 Memory Analysis - Volatility

The following resource contains additional descriptions of other injection techniques and their respective detection.

https://www.fireeye.com/blog/threat-research/2019/10/process-injection-techniques-technical-survey-common-and-trending-process

3.4.2 Memory Analysis - Volatility

Volatility has a plugin "yarascan" which allows you to search for strings, patterns, and also compound rules. As stated on its wiki page, this plugin can help you locate any sequence of bytes (like assembly instructions with wild cards), regular expressions, ANSI strings, or Unicode strings in user mode or kernel memory.

You can also use a YARA rules file as an argument instead of specifying the rule(s) on the command line.

https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#yarascan

3.4.2 Memory Analysis - Volatility

In some instances, you may be hunting for very sophisticated pieces of malware (rootkits) where you have to dig into system objects such as drivers, mutexes, and hooked functions.

3.4.2 Memory Analysis - Volatility

The following plugins are extremely helpful in this area:

- idt
- ssdt
- apihooks
- modules
- modscan
- driverirp
- driverscan

An example and walkthrough of rootkit detection is available here.

https://eforensicsmag.com/finding-advanced-malware-using-volatility/

3.4.2 Memory Analysis - Volatility

If any of the hunting activities identified a threat, Volatility provides a wide range of modules that will help you extract or rather, carve out of the memory dump all of the malicious object(s) (process, driver, ...) for further analysis.

3.4.2 Memory Analysis - Volatility

Further details on how to use Volatility are available on its Wiki page, here.

Lastly, if you want to play with some memory samples and perfect your Volatility knowledge, you can download them from here.

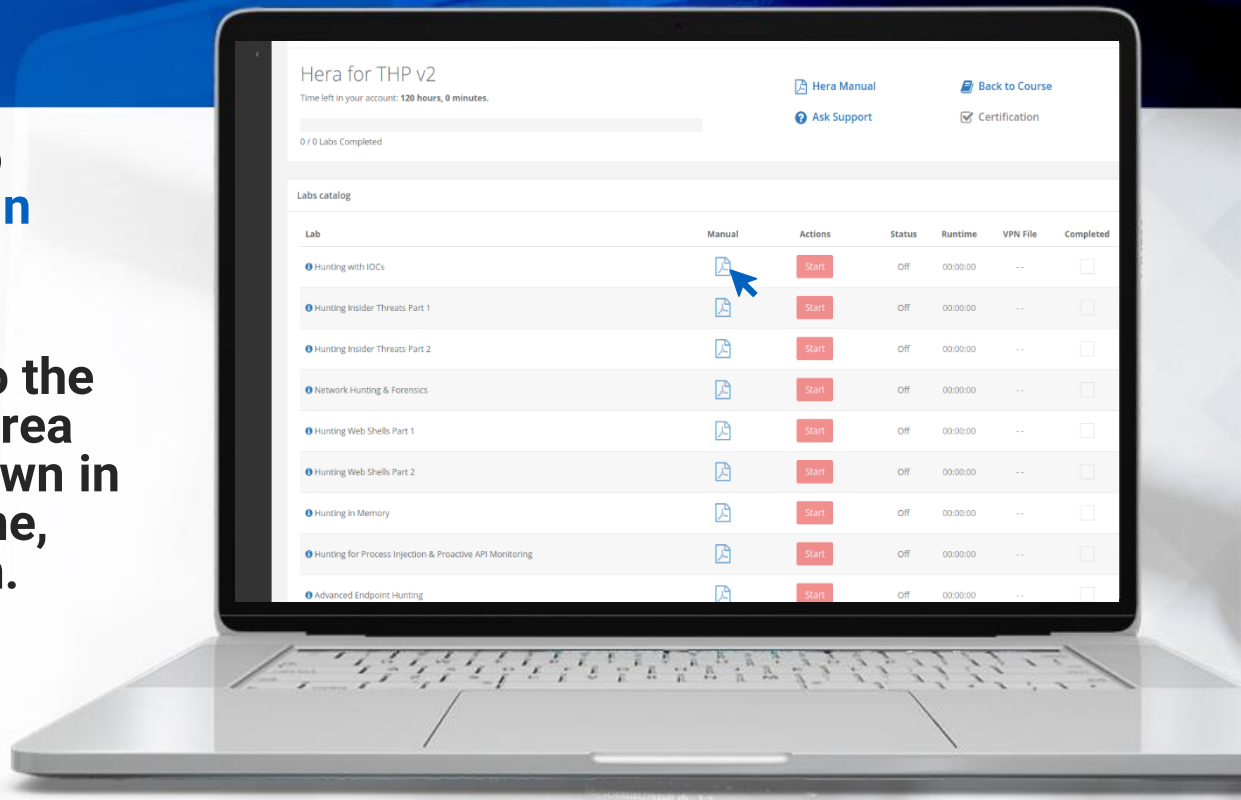
https://github.com/volatilityfoundation/volatility/wiki/Volatility-Usage
https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples

3.4.2.1 Hera Lab

Put what you've learned to practice with the **Hunting in Memory** lab!

To ACCESS your lab, go to the course in your members area and click the labs drop-down in the appropriate module line, then click the manual icon.

All labs are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.



***NOTE: some courses contain several labs and manuals, please make sure to click the file icon as it may be a zip that contains multiple lab manuals.**

3.4.3 Live System Memory Hunting

Unfortunately, getting a memory dump (from all your systems) and performing analysis on it is rather impractical. It is too time consuming, and therefore the hunts are performed on a subset of hosts only.

Another obstacle is the memory size – on average, the size is 16GB from workstations and commonly 64 GB (or more) on servers.

3.4.3 Live System Memory Hunting

Recently, some tools have emerged that attempt to scale memory hunting, primarily focusing on detecting injected code on the live machine without the need of obtaining memory dumps. The tools we'll look into are:

- GetInjectedThread.ps1
- Memhunter
- Captain

https://gist.github.com/jaredcatkinson/23905d34537ce4b5b1818c3e6405c1d2
https://github.com/marcosd4h/memhunter
https://github.com/y3n11/Captain

3.4.3.1 Live System Memory Hunting - Get-InjectedThread

Get-InjectedThread is defined by the author as a tool that can detect:

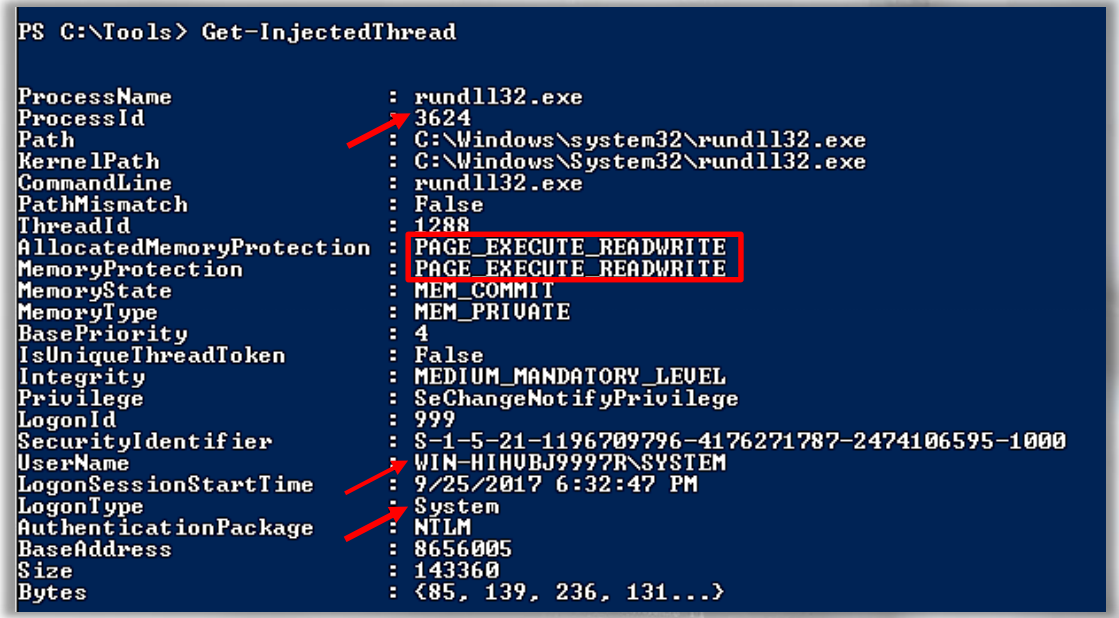
- Classic Injection
- Reflective DLL Injection
- Memory Module (similar technique to RDI)

The original presentation of the tool is available here.

https://www.usenix.org/cyber-security-summit/archives/file/summit-archive-1492714038.pdf

3.4.3.1 Live System Memory Hunting - Get-InjectedThread

Running the script on a compromised host returns confirmation of the injection and additional information about the process and thread detected.

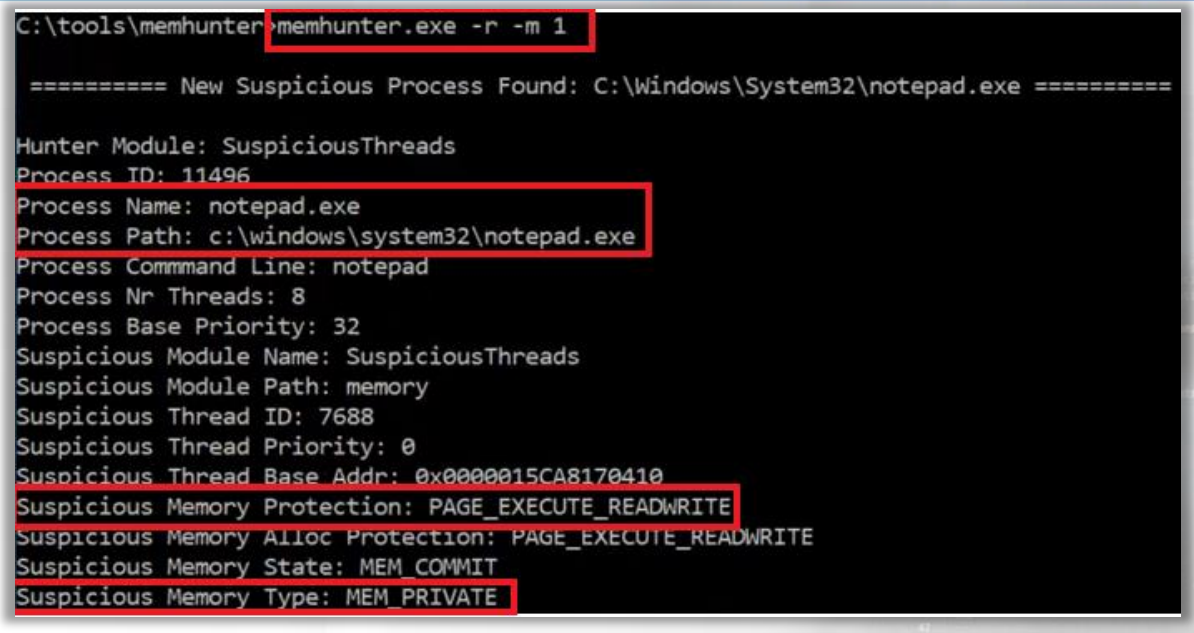


3.4.3.2 Live System Memory Hunting - Memhunter

Memhunter is a standalone binary that, upon execution, deploys itself as a Windows service.

Once installed, it feeds data to memory inspection scanners that use detection heuristics to locate potential attacks.

3.4.3.2 Live System Memory Hunting - Memhunter



A working PoC video of the tool is available here.

https://www.youtube.com/watch?v=t_fR1sCENkc

3.4.3.3 Live System Memory Hunting - Captain

Captain is an endpoint monitoring tool that is designed to spot malicious events through API hooking.

Captain, among others, is cable of detecting :

- Code Injection
- Memory dump creation (e.g. dump of LSASS)
- Fileless malware
- Execution of Office macros

3.4.3.3 Live System Memory Hunting - Captain

Captain requires its 4 components to operate:

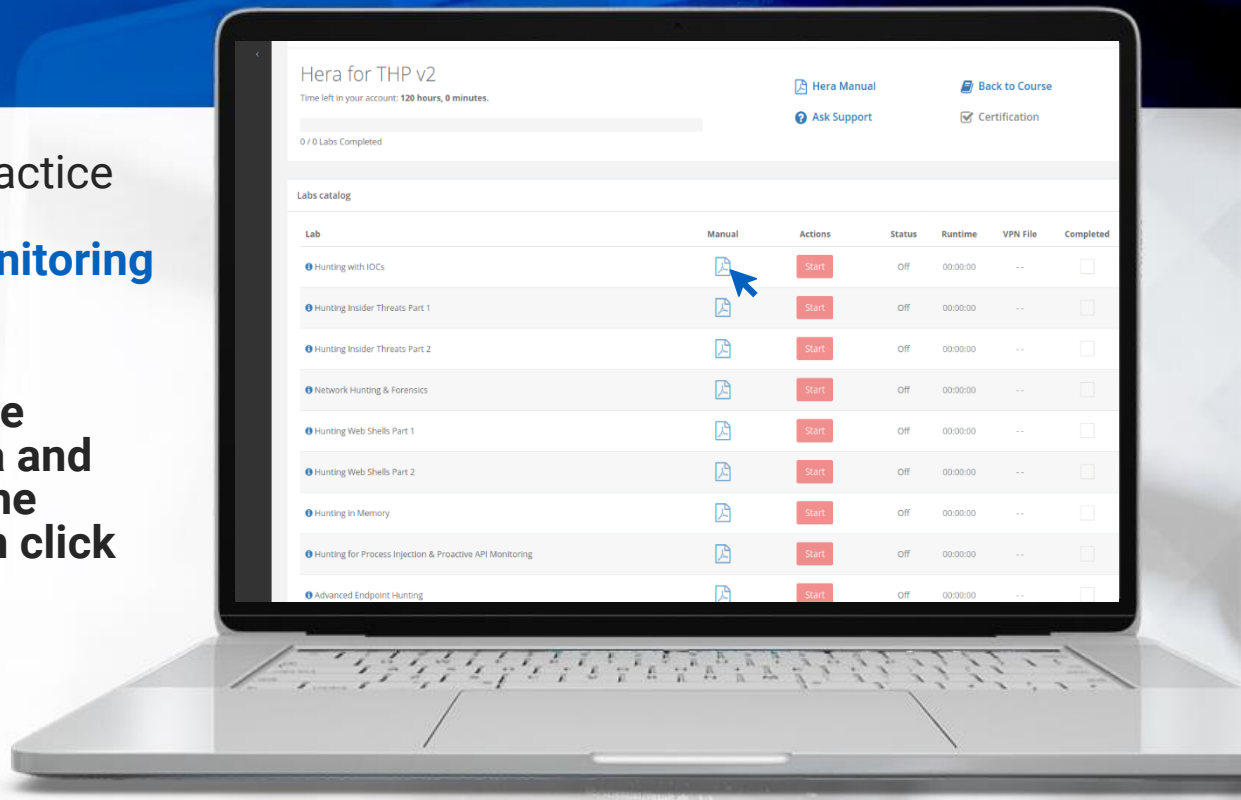
- **Monitor.ps1** – Monitors for process creations and injects Captain.dll in new processes
- **Injector.exe** – Used for the injection of Captain.dll
- **Captain.dll** – Hooks Windows API functions and outputs events
- **Behan.py** – analyzes the events captured by Captain.dll (based on provided signatures for alerting)

3.4.4 Hera Lab

Put what you've learned to practice with the **Hunting for Process Injection & Proactive API Monitoring** lab!

To ACCESS your lab, go to the course in your members area and click the labs drop-down in the appropriate module line, then click the manual icon.

All labs are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.



***NOTE: some courses contain several labs and manuals, please make sure to click the file icon as it may be a zip that contains multiple lab manuals.**

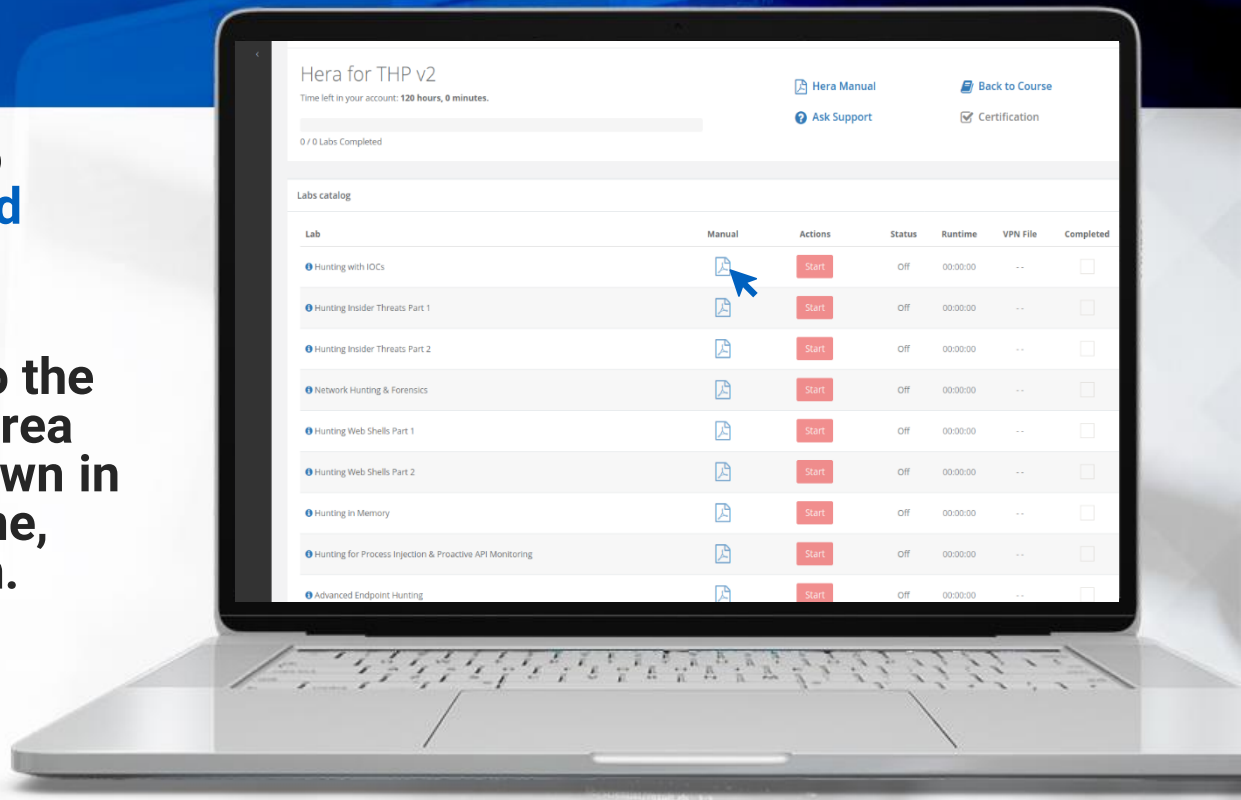
https://t.me/learningnets

3.4.5 Hera Lab

Put what you've learned to practice with the **Advanced Endpoint Hunting** lab!

To ACCESS your lab, go to the course in your members area and click the labs drop-down in the appropriate module line, then click the manual icon.

All labs are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.



***NOTE: some courses contain several labs and manuals, please make sure to click the file icon as it may be a zip that contains multiple lab manuals.**

Malware Analysis

3.5 Malware Analysis

Even though malware analysis is beyond the scope of this course, it's still worth a mention.

Malware analysis is needed when a binary needs to be analyzed further. We know that malware, whether it's packed, encrypted, etc., is in clear-text in memory, but in order to further understand the malware, analysis is needed.

3.5 Malware Analysis

If your security team doesn't have a dedicated malware analyst, then as a threat hunter, this is a skill to have. Even if it's basic malware analysis skills, it will be helpful.

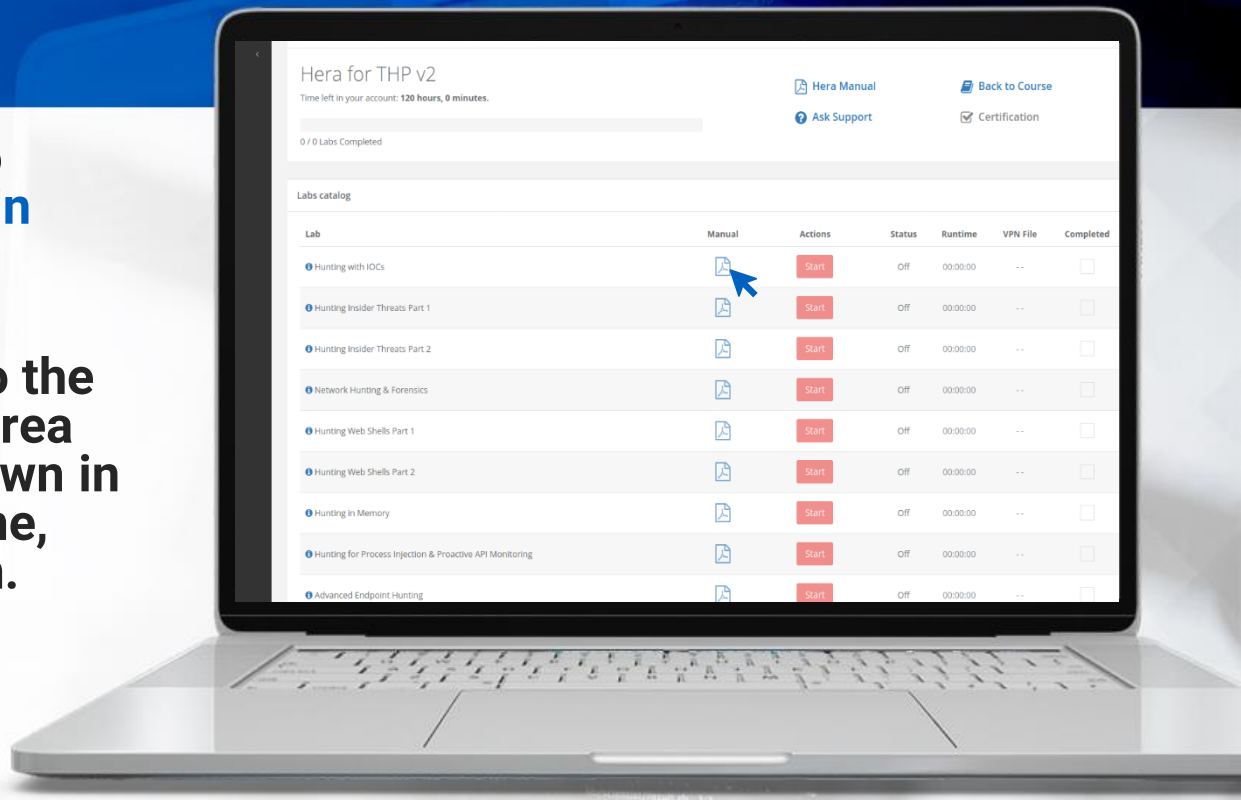
A threat hunter is similar to a spec ops operator. No matter what he/she encounters, he/she is trained and has the skill to complete the task. Whether it is inspecting network traffic, hunting for malicious files in various operating systems, performing incident response, memory analysis, etc., they are ready.

3.5.1 Hera Lab

Put what you've learned to practice with the **Hunting in Malware Part 1** lab!

To ACCESS your lab, go to the course in your members area and click the labs drop-down in the appropriate module line, then click the manual icon.

All labs are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.



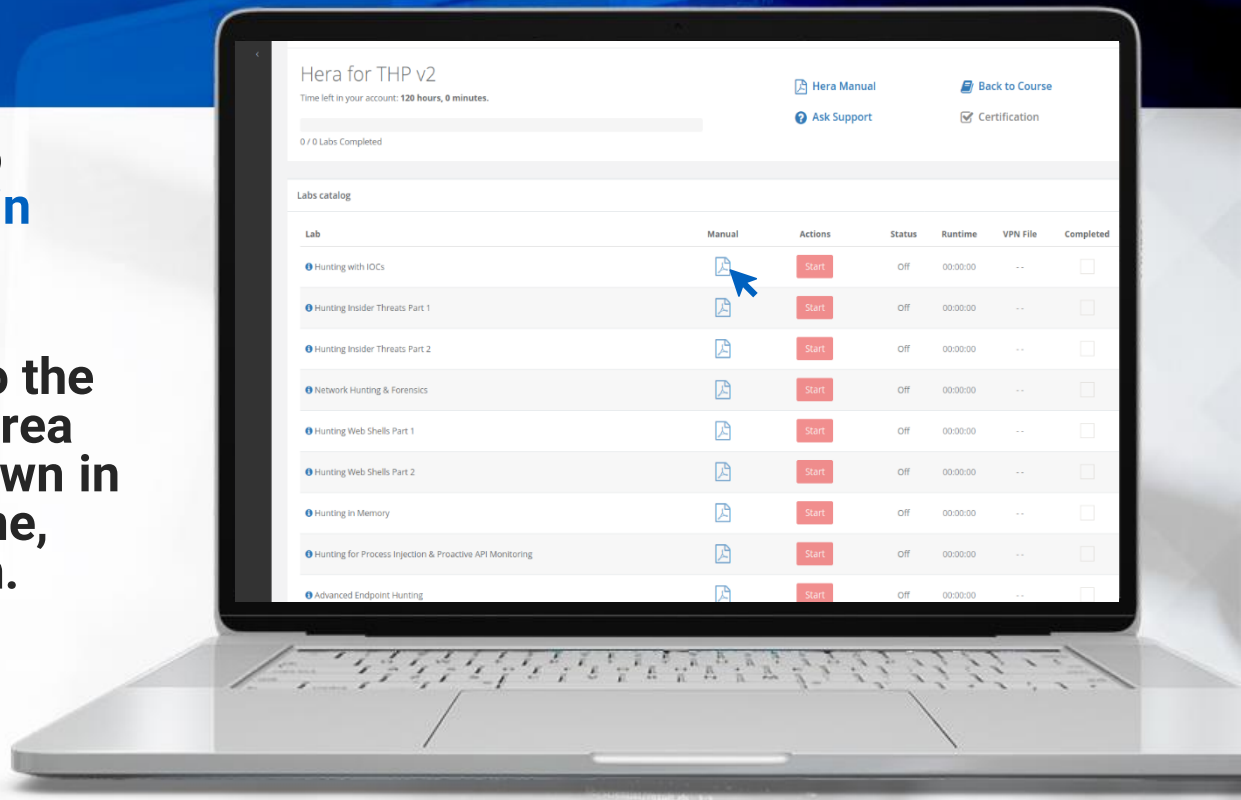
***NOTE: some courses contain several labs and manuals, please make sure to click the file icon as it may be a zip that contains multiple lab manuals.**

3.5.2 Hera Lab

Put what you've learned to practice with the **Hunting in Malware Part 2** lab!

To ACCESS your lab, go to the course in your members area and click the labs drop-down in the appropriate module line, then click the manual icon.

All labs are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.



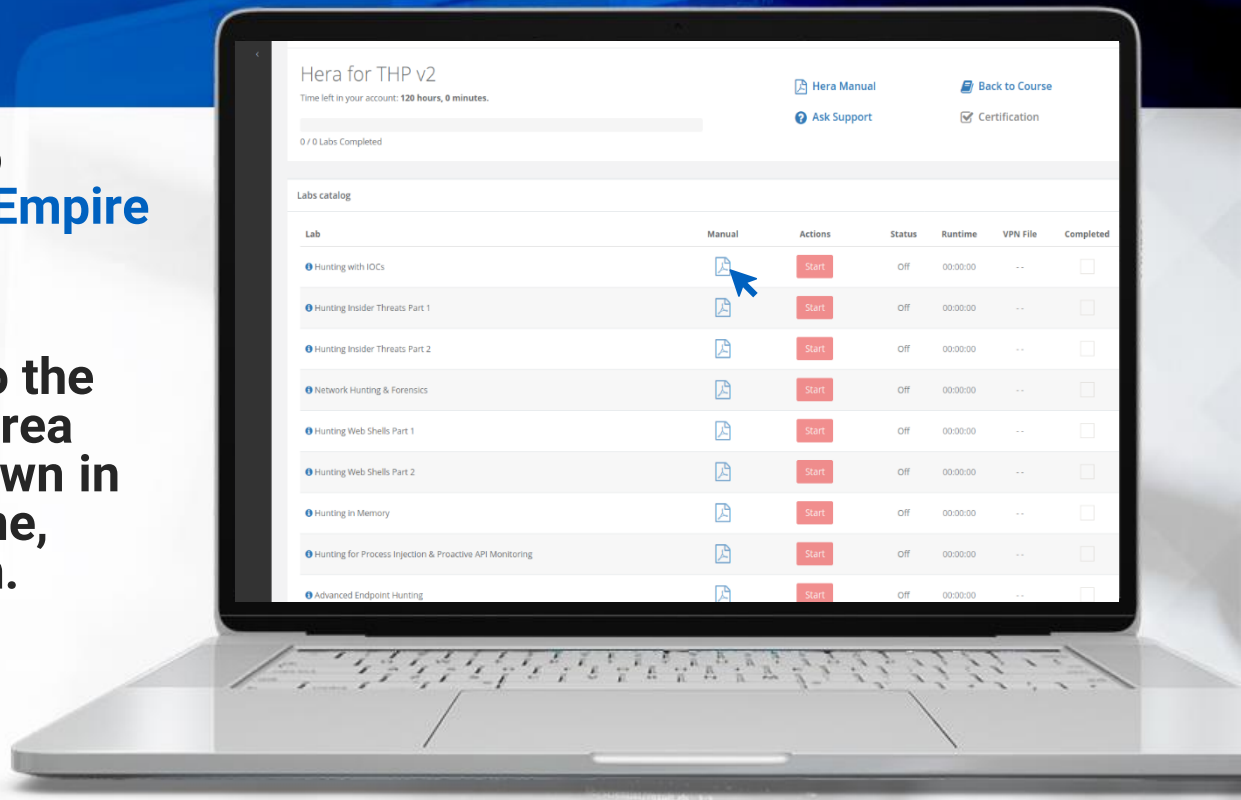
***NOTE: some courses contain several labs and manuals, please make sure to click the file icon as it may be a zip that contains multiple lab manuals.**

3.5.3 Hera Lab

Put what you've learned to practice with the **Hunting Empire** lab!

To ACCESS your lab, go to the course in your members area and click the labs drop-down in the appropriate module line, then click the manual icon.

All labs are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.



***NOTE: some courses contain several labs and manuals, please make sure to click the file icon as it may be a zip that contains multiple lab manuals.**

Conclusion

This concludes this module on Hunting Malware.

We have covered:

- ✓ Various detection tools
- ✓ Various detection techniques
- ✓ Memory analysis tools
- ✓ The importance of malware analysis

References

References

[PE Capture v1.2]

http://www.novirusthanks.org/products/pe-capture/

[PE Capture Service v1.2]

http://www.novirusthanks.org/products/pe-capture-service/

[RandomCode]

https://github.com/abhisek/RandomCode/tree/master/Malware/Process

[Meterpreter_Payload_Detection]

https://github.com/DamonMohammadbagher/Meterpreter_Payload_Detection

References

reflective-injection-detection

https://github.com/papadp/reflective-injection-detection

PowerShellArsenal

https://github.com/mattifestation/PowerShellArsenal

NtQueryInformationThread function

https://msdn.microsoft.com/en-us/library/windows/desktop/ms684283(v=vs.85).aspx

Get-InjectedThread.ps1

https://gist.github.com/jaredcatkinson/23905d34537ce4b5b1818c3e6405c1d2

References

ssdeep - Fuzzy hashing program

https://ssdeep-project.github.io/ssdeep/index.html

Identifying Almost Identical Files Using Context Triggered Piecewise Hashing

http://dfrws.org/sites/default/files/session-files/paper-identifying_almost_identical_files_using_context_triggered_piecewise_hashing.pdf

[How To] Fuzzy Hashing with SSDEEP (similarity matching)

https://dfir.science/2017/07/How-To-Fuzzy-Hashing-with-SSDEEP-(similarity-matching).html

ssdeep

https://github.com/ssdeep-project/ssdeep

References

VirusTotal += imphash

http://blog.virustotal.com/2014/02/virustotal-imphash.html

Tracking Malware with Import Hashing

https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html

ImpHash-Generator

https://github.com/Neo23x0/ImpHash-Generator

Caching Out: The Value of Shimcache for Investigators

https://www.fireeye.com/blog/threat-research/2015/06/caching_out_the_val.html

References

ShimCacheParser

https://github.com/mandiant/ShimCacheParser

Evolving Analytics for Execution Trace Data

https://www.fireeye.com/blog/threat-research/2017/04/appcompatprocessor.html

appcompatprocessor

https://github.com/mbevilacqua/appcompatprocessor

Reflective DLL Injection Detection through Memhunter

https://www.youtube.com/watch?v=t_fR1sCENkc

References

Hunting In Memory

https://www.sans.org/cyber-security-summit/archives/file/summit-archive-1492714038.pdf

FTK Imager

https://accessdata.com/product-download/ftk-imager-version-4-2-0

Comae Stardust

https://my.comae.com/

MAGNET RAM Capture

https://www.magnetforensics.com/resources/magnet-ram-capture/

References

Redline

https://www.fireeye.com/services/freeware/redline.html

volatility

https://github.com/volatilityfoundation/volatility

Captain

https://github.com/y3n11/Captain

memhunter

https://github.com/marcosd4h/memhunter

References

Volatility – Memory Samples

https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples

Volatility Usage

https://github.com/volatilityfoundation/volatility/wiki/Volatility-Usage

Malware Researcher's Handbook (Demystifying PE File)

https://resources.infosecinstitute.com/2-malware-researchers-handbook-demystifying-pe-file/#gref

DETECTING DECEPTIVE PROCESS HOLLOWING TECHNIQUES USING HOLLOWFIND VOLATILITY PLUGIN

https://cysinfo.com/detecting-deceptive-hollowing-techniques/

References

[Ten process injection techniques: A technical survey of common and trending process injection techniques]

https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

[yarascan]

https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#yarascan

[Finding Advanced Malware Using Volatility]

https://eforensicsmag.com/finding-advanced-malware-using-volatility/

Videos

Here's a list of all videos in this module. To **ACCESS** your video, go to the course in your members area and click the resources drop-down in the appropriate module line.

Note that all videos are only available in Full or Elite Editions of the course. To upgrade, click **LINK**.

Redline – Created Standard Collector

Redline – Basic Usage

Redline – Create Analysis File

Redline – Detecting Code Injection

Labs

Hunting in Memory

Lab 7.1: The organization you work for is asking you to perform memory threat hunting on a randomly selected machine. As a hunting exercise to keep you sharp, the IT Security manager tasked you specifically with looking for anomalous connections and memory injections.

Lab 7.2: The organization you work for is also asking you to perform memory threat hunting on a Linux machine. As a hunting exercise to keep you sharp, the IT Security manager tasked you specifically with looking for the existence of Linux rootkits.

Hunting for Process Injection & Proactive API Monitoring

Attackers love hiding/injecting malicious code into processes. In this lab, you will learn how to hunt for various process injection techniques and how to leverage userland API monitoring for more effective hunts.

Labs are only available in Full or Elite Editions of the course. To ACCESS your labs, go to the course in your members area and click the labs drop-down in the appropriate module line. To UPGRADE to gain access, click LINK.

Labs

Advanced Endpoint Hunting

Inside THP you will find two (2) distinct labs on advanced hacking techniques hunting at the endpoint level. Specifically, you will learn how to hunt for process doppelganging, AMSI bypasses, parent PID spoofing, reflective DLL injection, module stomping etc.

Hunting Malware Part 1

Your manager, Tony, wants you to keep an eye on the machine for the administrative assistant to the CFO. Email logs show that there has been a spike in spam emails attempting to reach her email address. Even though she has completed the security awareness class, Tony doesn't want to take any chances. Tony hands you a Mandiant Analysis File to load into Redline and see if there is anything suspicious that is running, or was running, on her machine. After analysis, Tony, requires you to get a recent Mandiant Analysis File to analyze.

Labs are only available in Full or Elite Editions of the course. To ACCESS your labs, go to the course in your members area and click the labs drop-down in the appropriate module line. To UPGRADE to gain access, click LINK.

Labs

Hunting Malware Part 2

Your manager, Tony, received 2 memory files from another facility within the ISAC. These 2 memory files were from actual incidents that took place within their facility a few years ago. Tony wants you to analyze them to see if you are able to analyze them for any signs of code injection and/or a rootkit to prepare you to detect APT attacks.

Hunting Empire

Your manager, Tony, wants to make sure that you can detect the widely used attacking tool, Empire. A hunting exercise has been scheduled, where you are tasked with detecting Empire's presence on an endpoint.

Labs are only available in Full or Elite Editions of the course. To ACCESS your labs, go to the course in your members area and click the labs drop-down in the appropriate module line. To UPGRADE to gain access, click LINK.