

YARA & Sigma for SOC Analysts

Introduction to YARA & Sigma

Overview

- YARA and Sigma are two essential tools used by SOC analysts to enhance their threat detection and incident response capabilities.
 - They empower analysts with improved threat detection capabilities, efficient log analysis, malware detection and classification, IOC identification, collaboration, customization, and integration with existing security tools.
- Both YARA and Sigma rules grant SOC analysts potent capabilities to detect and respond to security threats.
 - YARA excels in file and memory analysis, as well as pattern matching, whereas Sigma is particularly adept at log analysis and SIEM systems.
- These detection rules utilize conditional logic applied to logs or files.
 - Analysts craft these rules to pinpoint suspicious activities in logs or match patterns in files.
 - These rules are pivotal in making detections more straightforward to compose, and thus, they constitute a crucial element of an effective threat detection strategy.
 - Both YARA and Sigma adhere to standard formats that facilitate the creation and sharing of detection rules within the cybersecurity community.

Importance of YARA and Sigma rules for SOC Analysts

Let's explore the key reasons why YARA and Sigma are invaluable for SOC analysts:

- Enhanced Threat Detection: YARA and Sigma rules allow SOC analysts to develop customized detection rules tailored to their unique environment and security needs.
 - These rules empower analysts to discern patterns, behaviors, or indicators linked to security threats, thus enabling them to proactively detect and address potential incidents.
 - Various Github repositories provide a wealth of examples of YARA and Sigma rules.
 - YARA rules : <https://github.com/Yara-Rules/rules/tree/master/malware>,
<https://github.com/mikesxrs/Open-Source-YARA-rules/tree/master>

- **Sigma rules** <https://github.com/SigmaHQ/sigma/tree/master/rules>,
<https://github.com/joesecurity/sigma-rules>,
<https://github.com/mdecrevoisier/SIGMA-detection-rules>
- **Efficient Log Analysis**: Sigma rules are essential for log analysis in a SOC setting.
 - Utilizing Sigma rules, analysts can filter and correlate log data from disparate sources, concentrating on events pertinent to security monitoring.
 - This minimizes irrelevant data and enables analysts to prioritize their investigative efforts, leading to more efficient and effective incident response.
 - An open-source tool called [Chainsaw](#) can be used to apply Sigma rules to event log files.
- **Collaboration and Standardization**: YARA and Sigma offer standardized formats and rule structures, fostering collaboration among SOC analysts and tapping into the collective expertise of the broader cybersecurity community.
 - This encourages knowledge sharing, the formulation of best practices, and keeps analysts abreast of cutting-edge threat intelligence and detection methodologies.
 - For instance, "The DFIR Report" shares YARA and Sigma rules derived from their investigations.
 - <https://github.com/The-DFIR-Report/Yara-Rules>
 - <https://github.com/The-DFIR-Report/Sigma-Rules>
- **Integration with Security Tools**: YARA and Sigma rules can be integrated seamlessly with a plethora of security tools, including SIEM platforms, log analysis systems, and incident response platforms.
 - This integration enables automation, correlation, and enrichment of security events, allowing SOC analysts to incorporate the rules into their existing security infrastructure.
 - As an example, [Uncoder.io](#) facilitates the conversion of Sigma rules into tailor-made, performance-optimized queries ready for deployment in the chosen SIEM and XDR systems.

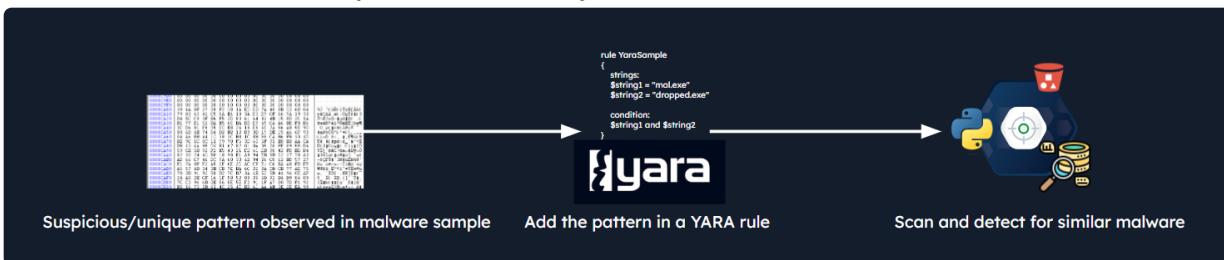
Leveraging YARA

YARA and YARA Rules

Overview

- **YARA** is a powerful pattern-matching tool and rule format used for identifying and classifying files based on specific patterns, characteristics, or content.

- SOC analysts commonly use YARA rules to detect and classify malware samples, suspicious files, or indicators of compromise (IOCs).
- YARA rules are typically written in a rule syntax that defines the conditions and patterns to be matched within files.
 - These rules can include various elements, such as strings, regular expressions, and Boolean logic operators, allowing analysts to create complex and precise detection rules.
 - It's important to note that YARA rules can recognize both textual and binary patterns, and they can be applied to memory forensics activities as well.
- When applied, YARA scans files or directories and matches them against the defined rules.
 - If a file matches a specific pattern or condition, it can trigger an alert or warrant further examination as a potential security threat.



- YARA rules are especially useful for SOC analysts when analyzing malware samples, conducting forensic investigations, or performing threat hunting activities.
 - The flexibility and extensibility of YARA make it a valuable tool in the cybersecurity community.

Developing YARA Rules

Overview

- In this section, we'll cover manual and automated YARA rule development.
- Let's dive into the world of YARA rules using a sample named `svchost.exe` residing in the `/home/htb-student/Samples/YARASigma` directory of this section's target as an illustration.
 - We want to understand the process behind crafting a YARA rule, so let's get our hands dirty.
- Initially, we need to conduct a string analysis on our malware sample.

```
areaeric@htb[/htb]$ strings svchost.exe
```

```
areaeric@htb[/htb]$ strings svchost.exe
!This program cannot be run in DOS mode.
UPX0
UPX1
UPX2
3.96
UPX!
8MZu
HcP<H
<1~o
VDgxt
D$ /
OAUATUWVSH
15384
[^_ ]A\A]
^$<V
---SNIP---
X]_^[H
QRAPH
(AXZY
KERNEL32.DLL
msvcrt.dll
ExitProcess
GetProcAddress
LoadLibraryA
VirtualProtect
exit
```

- From the first few strings, it becomes evident that the file is packed using the UPX (Ultimate Packer for eXecutables) packer.
 - Given this discovery, we can incorporate UPX-related strings to formulate a basic

YARA rule targeting samples packed via UPX.

Code: `yara`

```
rule UPX_packed_executable
{
    meta:
        description = "Detects UPX-packed executables"

    strings:
        $string_1 = "UPX0"
        $string_2 = "UPX1"
        $string_3 = "UPX2"

    condition:
        all of them
}
```

- Here's a brief breakdown of our YARA rule crafted for detecting UPX-packed executables:
 - Rule Name : `UPX_packed_executable`
 - Meta Description : Provides a description of the rule, stating that it detects UPX-packed executables.
 - Strings Section: `strings` defines the strings that the rule will search for within the files.
 - `$string_1 = "UPX0"`: Matches the string `UPX0` within the file.
 - `$string_2 = "UPX1"`: Matches the string `UPX1` within the file.
 - `$string_3 = "UPX2"`: Matches the string `UPX2` within the file.
 - Condition : `condition` specifies the criteria that must be met for the rule to trigger a match.
 - `all of them`: Specifies that all the defined strings (`$string_1`, `$string_2`, and `$string_3`) must be found within the file.
- In essence, our `UPX_packed_executable` rule (located inside this section's target at `/home/htb-student/Rules/yara/upx_packed.yar`) scans for the strings `UPX0`, `UPX1`, and `UPX2` inside a file. I
 - If the rule finds all three strings, it raises an alert, hinting that the file might be packed with the UPX packer.
 - This rule is a handy tool when we're on the lookout for executables that have undergone compression or obfuscation using the UPX method.

Developing a YARA Rule Through yarGen

- Let's continue our dive into the world of YARA rules using a sample named `dharma_sample.exe` residing in the `/home/htb-student/Samples/YARASigma` directory of this section's target.
 - Once again, we need to conduct a string analysis on our malware sample.

```
areaeric@htb[~/htb]$ strings dharma_sample.exe
```

- After we execute the `strings` command on `dharma_sample.exe`, we spot `C:\crysis\Release\PDB\payload.pdb`, which is pretty unique.
 - Alongside other distinct strings, we can craft a more refined YARA rule. Let's employ `yarGen` to expedite this process.
 - `yarGen` is our go-to tool when we need an automatic YARA rule generator.
 - What makes it a gem is its ability to churn out YARA rules based on strings found in malicious files while sidestepping strings common in benign software.
 - This is possible because `yarGen` comes equipped with a vast database of goodware strings and opcodes.

- Before diving in, we need to unpack the ZIP archives containing these databases.
- Let's place our sample in a `temp` directory (there is one available at `/home/htb-student/temp` inside this section's target) and specify the path using the following command-line arguments.

```
areaeric@htb[/htb]$ python3 yarGen.py -m /home/htb-student/temp -o htb_sample.yar
```

```
areaeric@htb[htb]$ python3 yarGen.py -m /home/htb-student/temp -o htb_sample.yar
-----
  _ _ _ _ _ / _/_/ _ _ 
 / // / _ `/_/(_/_/_)_\ 
 \_, \_/_/_/ \_/_/\_/_//_/
 /_/_/ Yara Rule Generator
 Florian Roth, July 2020, Version 0.23.3

Note: Rules have to be post-processed
See this post for details: https://medium.com/@cyb3rops/121d29322282
-----
[+] Using identifier 'temp'
[+] Using reference 'https://github.com/Neo23x0/yarGen'
[+] Using prefix 'temp'
[+] Processing PEStudio strings ...
[+] Reading goodware strings from database 'good-strings.db' ...
  (This could take some time and uses several Gigabytes of RAM depending on your db size)
[+] Loading ./dbs/good-imphashes-part3.db ...
[+] Total: 4029 / Added 4029 entries
[+] Loading ./dbs/good-strings-part9.db ...
[+] Total: 788 / Added 788 entries
[+] Loading ./dbs/good-strings-part8.db ...
[+] Total: 332082 / Added 331294 entries
[+] Loading ./dbs/good-imphashes-part4.db ...
[+] Total: 6426 / Added 2397 entries
[+] Loading ./dbs/good-strings-part2.db ...
[+] Total: 1703601 / Added 1371519 entries
[+] Loading ./dbs/good-exports-part2.db ...
[+] Total: 90960 / Added 90960 entries
[+] Loading ./dbs/good-strings-part4.db ...
[+] Total: 3860655 / Added 2157054 entries
[+] Loading ./dbs/good-exports-part4.db ...
[+] Total: 172718 / Added 81758 entries
```

Command Breakdown:

- `yarGen.py` : This is the name of the yarGen Python script that will be executed.
- `-m /home/htb-student/temp` : This option specifies the source directory where the sample files (e.g., malware or suspicious files) are located.
 - The script will analyze these samples to generate YARA rules.
- `-o htb_sample.yar` : This option indicates the output file name for the generated YARA rules. In this case, the YARA rules will be saved to a file named

`htb_sample.yar`.

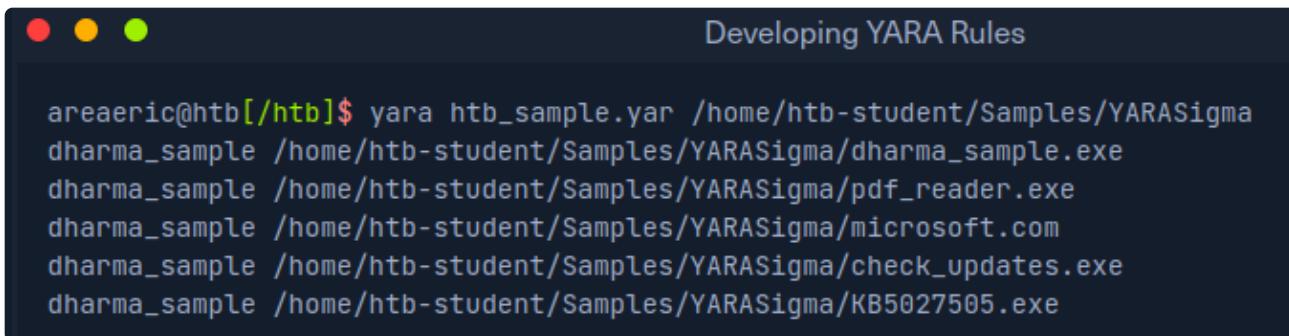
- The resulting YARA rules will be written to the `htb_sample.yar` file inside the `/home/htb-student/yarGen-0.23.4` directory of this section's target.
 - Let's see the content of the generated rule.

```
areaeric@htb[/htb]$ cat htb_sample.yar
```

```
rule dharma_sample {
    meta:
        description = "temp - file dharma_sample.exe"
        author = "yarGen Rule Generator"
        reference = "https://github.com/Neo23x0/yarGen"
        date = "2023-08-24"
        hash1 = "bfff6a1000a86f8edf3673d576786ec75b80bed0c458a8ca0bd52d12b74099071"
    strings:
        $x1 = "C:\\\\crysis\\\\Release\\\\PDB\\\\payload.pdb" fullword ascii
        $s2 = "ssssssbs" fullword ascii
        $s3 = "ssssssbsss" fullword ascii
        $s4 = "RSDS%~m" fullword ascii
        $s5 = "{RDqP^\\\\\" fullword ascii
        $s6 = "QtVN$0w" fullword ascii
        $s7 = "Ffsc<{" fullword ascii
        $s8 = "^N3Y.H_K" fullword ascii
        $s9 = "tb#w\\\\6" fullword ascii
        $s10 = "-j6EPUC" fullword ascii
        $s11 = "8QS#5@3" fullword ascii
        $s12 = "h1+LI;d8" fullword ascii
        $s13 = "H;B cl" fullword ascii
        $s14 = "Wy]z@p]E" fullword ascii
        $s15 = "ipgypA" fullword ascii
        $s16 = "+>^wI{H" fullword ascii
        $s17 = "mF@S/]" fullword ascii
        $s18 = "OA_<8X-|" fullword ascii
        $s19 = "s+aL%M" fullword ascii
        $s20 = "sXtY9P" fullword ascii
    condition:
        uint16(0) == 0x5a4d and filesize < 300KB and
        1 of ($x*) and 4 of them
}
```

- Now, for the moment of truth.
 - We'll unleash YARA with our newly minted rule to see if it sniffs out any matches when run against a malware sample repository located at `/home/htb-student/Samples/YARASigma` inside this section's target.

```
areaeric@htb[/htb]$ yara htb_sample.yar /home/htb-student/Samples/YARASigma
```



```
areaeric@htb[/htb]$ yara htb_sample.yar /home/htb-student/Samples/YARASigma  
dharma_sample /home/htb-student/Samples/YARASigma/dharma_sample.exe  
dharma_sample /home/htb-student/Samples/YARASigma/pdf_reader.exe  
dharma_sample /home/htb-student/Samples/YARASigma/microsoft.com  
dharma_sample /home/htb-student/Samples/YARASigma/check_updates.exe  
dharma_sample /home/htb-student/Samples/YARASigma/KB5027505.exe
```

- As we can see, the `pdf_reader.exe`, `microsoft.com`, `check_updates.exe`, and `KB5027505.exe` files are detected by this rule (in addition to `dharma_sample.exe` of course).

Manually Developing a YARA Rule

Example 1: ZoxPNG RAT Used by APT17

- Let's now go a bit deeper...
- We want to develop a YARA rule to scan for a specific variation of the `ZoxPNG` RAT used by `APT17` based on:
 - A sample named `legit.exe` residing in the `/home/htb-student/Samples/YARASigma` directory of this section's target
 - A [post from Intezer](#)
 - String analysis
 - Imphash
 - Common sample file size
- Let's start with our string analysis endeavors as follows.

```
areaeric@htb[/htb]$ strings legit.exe
```



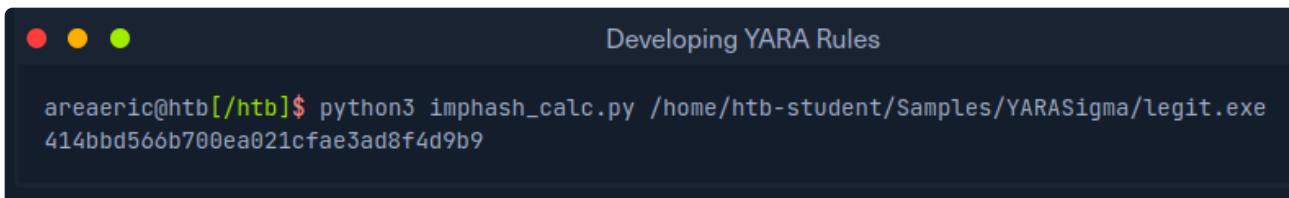
```
areaeric@htb[/htb]$ strings legit.exe
!This program cannot be run in DOS mode.
Rich
.text
`.rdata
@.data
---SNIP---
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
deflate 1.1.4 Copyright 1995-2002 Jean-loup Gailly

inflate 1.1.4 Copyright 1995-2002 Mark Adler
Sleep
LocalAlloc
CloseHandle
GetLastError
VirtualFree
VirtualAlloc
GetProcAddress
LoadLibraryA
GetCurrentProcessId
GlobalMemoryStatusEx
GetCurrentProcess
GetACP
GetVersionExA
GetComputerNameA
GetTickCount
GetSystemTime
LocalFree
CreateProcessA
CreatePipe
TerminateProcess
ReadFile
PeekNamedPipe
WriteFile
SetFilePointer
CreateFileA
GetFileSize
GetDiskFreeSpaceExA
```

- Let's then use the hashes mentioned in Intezer's post to identify common sample sizes.
 - It looks like there are no related samples whose size is bigger than 200KB.
 - An example of an identified sample is the following. <https://www.hybrid-analysis.com/sample/ee362a8161bd442073775363bf5fa1305abac2ce39b903d63df0d7121ba60550>.

- Finally, the sample's Imphash can be calculated as follows, using the `imphash_calc.py` script that resides in the `/home/htb-student` directory of this section's target.

```
areaeric@htb[/htb]$ python3 imphash_calc.py /home/htb-
student/Samples/YARASigma/legit.exe
```



The screenshot shows a terminal window with three colored window control buttons (red, yellow, green) at the top. The title bar reads "Developing YARA Rules". The main area of the terminal contains the following text:

```
areaeric@htb[/htb]$ python3 imphash_calc.py /home/htb-student/Samples/YARASigma/legit.exe
414bbd566b700ea021cfaf3ad8f4d9b9
```

- A good YARA rule to detect the aforementioned variation of ZoxPNG resides in the `/home/htb-student/Rules/yara` directory of this section's target, saved as `apt_apt17_mal_sep17_2.yar`.

```
/*
Yara Rule Set
Author: Florian Roth
Date: 2017-10-03
Identifier: APT17 Oct 10
Reference: https://goo.gl/puVc9q
*/

/* Rule Set -----
----- */

import "pe"

rule APT17_Malware_Oct17_Gen {
    meta:
        description = "Detects APT17 malware"
        license = "Detection Rule License 1.1
https://github.com/Neo23x0/signature-base/blob/master/LICENSE"
        author = "Florian Roth (Nextron Systems)"
        reference = "https://goo.gl/puVc9q"
        date = "2017-10-03"
        hash1 =
"0375b4216334c85a4b29441a3d37e61d7797c2e1cb94b14cf6292449fb25c7b2"
        hash2 =
"07f93e49c7015b68e2542fc591ad2b4a1bc01349f79d48db67c53938ad4b525d"
        hash3 =
```

```

"ee362a8161bd442073775363bf5fa1305abac2ce39b903d63df0d7121ba60550"
  strings:
    $x1 = "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64;
Trident/4.0; SLCC2; .NETCLR 2.0.50727)" fullword ascii
    $x2 = "http://%s/imgres?q=A380&hl=en-
US&sa=X&biw=1440&bih=809&tbo=ius&tbnid=aLW4-J8Q1lmYBM" ascii

    $s1 = "hWritePipe2 Error:%d" fullword ascii
    $s2 = "Not Support This Function!" fullword ascii
    $s3 = "Cookie: SESSIONID=%s" fullword ascii
    $s4 = "http://0.0.0.0/1" fullword ascii
    $s5 = "Content-Type: image/x-png" fullword ascii
    $s6 = "Accept-Language: en-US" fullword ascii
    $s7 = "IISCMD Error:%d" fullword ascii
    $s8 = "[IISEND=0x%08X][Recv:] 0x%08X %s" fullword ascii
  condition:
    ( uint16(0) == 0x5a4d and filesize < 200KB and (
      pe.imphash() == "414bbd566b700ea021cfae3ad8f4d9b9" or
      1 of ($x*) or
      6 of them
    )
  )
}

```

YARA Rule Breakdown:

- **Rule Imports :** Modules are extensions to YARA's core functionality.
 - `import "pe"` : By importing the **PE module** the YARA rule gains access to a set of specialized functions and structures that can inspect and analyze the details of **PE** files.
 - This makes the rule more precise when it comes to detecting characteristics in Windows executables.
- **Rule Meta :**
 - `description` : Tells us the main purpose of the rule, which is to detect APT17 malware.
 - `license` : Points to the location and version of the license governing the use of this YARA rule.
 - `author` : The rule was written by Florian Roth from Nextron Systems.
 - `reference` : Provides a link that goes into more detail about the malware or context of this rule.
 - `date` : The date the rule was either created or last updated, in this case, 3rd October 2017.

- `hash1`, `hash2`, `hash3`: Hash values, probably of samples related to APT17, which the author used as references or as foundational data to create the rule.
- **Rule Body**: The rule contains a series of strings, which are potential indicators of the APT17 malware.
 - These strings are split into two categories
 - `$x*` strings
 - `$s*` strings
- **Rule Condition**: This is the heart of the rule, where the actual detection logic resides.
 - `uint16(0) == 0x5a4d`: Checks if the first two bytes of the file are `MZ`, which is the magic number for Windows executables.
 - So, we're focusing on detecting Windows binaries.
 - `filesize < 200KB`: Limits the rule to scan only small files, specifically those smaller than `200KB`.
 - `pe.imphash() == "414bbd566b700ea021cfac3ad8f4d9b9"`: This checks the import hash (`imphash`) of the PE (Portable Executable) file.
 - ImpHashes are great for categorizing and clustering malware samples based on the libraries they import.
 - `1 of ($x*)`: At least `one` of the `$x` strings (from the strings section) must be present in the file.
 - `6 of them`: Requires that at least `six` of the strings (from both `$x` and `$s` categories) be found within the scanned file.

Example 2: Neuron Used by Turla

- We want to develop a YARA rule to scan for instances of `Neuron Service` used by `Turla` based on:
 - A sample named `Microsoft.Exchange.Service.exe` residing in the `/home/htb-student/Samples/YARASigma` directory of this section's target
 - An [analysis report from the National Cyber Security Centre](#)
- Since the report mentions that both the Neuron client and Neuron service are written using the .NET framework we will perform .NET "reversing" instead of string analysis.
- This can be done using the `monodis` tool as follows.

```
areaeric@htb[/htb]$ monodis --output=code Microsoft.Exchange.Service.exe
```

```
areaeric@htb[/htb]$ cat code
```

```

areaeric@htb[/htb]$ cat code
.assembly extern System.Configuration.Install
{
    .ver 4:0:0:0
    .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A ) // .?_....:
}
---SNIP---
.class public auto ansi abstract sealed beforefieldinit StorageUtils
} // end of class Utils.StorageUtils
---SNIP---
        default void ExecCMD (string path, string key, unsigned int8[] cmd, class Utils.Config cfg, c
    IL_0028: ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscorlib]S
    IL_0070: stsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscorlib]S
    IL_0075: ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscorlib]S
    IL_007f: ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscorlib]S
} // end of method Storage::ExecCMD
.class nested private auto ansi abstract sealed beforefieldinit '<ExecCMD>o__SiteContainer0'
} // end of class <ExecCMD>o__SiteContainer0
    IL_0077: call void class Utils.Storage::ExecCMD(string, string, unsigned int8[], class Utils.C
---SNIP---
    IL_0029: ldftn void class Utils.Storage::KillOldThread()
        default void KillOldThread () cil managed
    } // end of method Storage::KillOldThread
---SNIP---

    IL_0201: ldstr "EncryptScript"
    IL_04a4: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[]
    IL_0eff: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[]
    IL_0f4e: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
    IL_0fec: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
    IL_102f: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
    IL_0018: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
    IL_00b1: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
        IL_009a: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
        IL_0142: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
            default unsigned int8[] EncryptScript (unsigned int8[] pwd, unsigned int8[] data) cil managed
    } // end of method Crypt::EncryptScript
        IL_00a0: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
        IL_0052: call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned int8[])
---SNIP---

```

- By going through the above we can identify functions and classes within the .NET assembly.

Note: A better reversing solution would be to load the .NET assembly (`Microsoft.Exchange.Service.exe`) into a .NET debugger and assembly editor like

dnSpy.

The screenshot shows the Microsoft Visual Studio interface with the 'Assembly Explorer' and 'Storage.cs' code editor tabs selected.

Assembly Explorer:

- mscorlib (4.0.0.0)
- System (4.0.0.0)
- System.Core (4.0.0.0)
- System.Xml (4.0.0.0)
- System.Xml.Linq (4.0.0.0)
- WindowsBase (4.0.0.0)
- PresentationCore (4.0.0.0)
- PresentationFramework (4.0.0.0)
- WPF (3.5.0.0)
- di Spy (6.4.0.0)
- System.Configuration (4.0.0.0)
- di Spy Contracts (6.4.0.0)
- Microsoft.VisualStudio.TextManager.Interop (15.0.0)
- Microsoft.VisualStudio.TextManagerLogic (15.0.0)
- Microsoft.VisualStudio.TextUI (15.0.0)
- Microsoft.VisualStudio.LanguageService (4.0.0.0)
- Microsoft.VisualStudio.CoreUtility (15.0.0)
- Microsoft.VisualStudio.TextData (15.0.0)
- Microsoft.VisualStudio.Composition (4.0.0.0)
- WindowsFormsIntegration (4.0.0.0)
- PresentationFramework.Aero (4.0.0.0)
- iCSSharpCode.TreeView (4.2.0.8752)
- System.ComponentModel.Composition
- Microsoft.Web.Hmac (4.0.0.0)
- PresentationFramework.Classic (4.0.0.0)
- PresentationUtil (4.0.0.0)
- Microsoft.Exchange.Service (1.0.0)
- Microsoft.Exchange.Service.exe

Storage.cs:

```
1  using System;
2  using System.Collections.Generic;
3  using System.IO;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using System.Threading;
7  using System.Web.Helpers;
8  using Microsoft.CSharp.RuntimeBinder;
9
10 namespace Util
11 {
12     // Token: 0x02000009 RID: 9
13     internal class Storage
14     {
15         // Token: 0x02000016 RID: 22 RVA: 0x0000037B8 File Offset: 0x000019B8
16         public static bool WaitAll(int timeout)
17         {
18             return Storage.listWait == null || Storage.listWait.Count <= 0 || WaitHandle.WaitAll(Storage.listWait.ToArray(), timeout);
19         }
20
21         // Token: 0x02000017 RID: 23 RVA: 0x0000037E0 File Offset: 0x000019E0
22         public static void KillOldThread()
23         {
24             for (;;)
25             {
26                 Storage.KillOld(null);
27                 Thread.Sleep(30000);
28             }
29         }
30
31         // Token: 0x02000018 RID: 24 RVA: 0x0000037F0 File Offset: 0x000019F0
32         private static void ExecCMD(string path, string key, byte[] cmd, Config cfg, ManualResetEvent mre)
33         {
34             FileStream fileStream = File.Create(path);
35             object obj = Json.Decode(Encoding.UTF8.GetString(Crypt.EncryptScript(Encoding.UTF8.GetBytes(key), cmd)));
36             if (Storage.<>c__DisplayClass0_0<SiteContainer>.<>p__Site1 == null)
37             {
38                 Storage.<>c__DisplayClass0_0<SiteContainer>.<>p__Site1 = CallSite.<Func<CallSite, Type, object, Config, CommandScript>.<Create>((bindn.InvokeConstructor(CSharpBinderFlags.None, typeof(Storage)), new CSharpArgumentInfo[0]
39
40                 {
41                     CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.UseCompileTimeType | CSharpArgumentInfoFlags.IsStaticType, null),
42                     CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.UseCompileTimeType, null),
43                     CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.UseCompileTimeType, null),
44                     CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.UseCompileTimeType, null)
45                 }));
46
47                 CommandScript commandScript = Storage.<>c__DisplayClass0_0<SiteContainer>.<>p__Site1.Target(Storage.<>c__DisplayClass0_0<SiteContainer>.<>p__Site1, typeof(CommandScript), obj, cfg);
48                 byte[] array = Crypt.EncryptScript(Encoding.UTF8.GetBytes(key), Encoding.UTF8.GetBytes(Json.Encode(commandScript)));
49                 fileStream.Write(array, 0, array.Length);
50                 fileStream.Close();
51                 mre.Set();
52             }
53
54             // Token: 0x02000019 RID: 25 RVA: 0x0000037F8 File Offset: 0x000019F8
55             public static void KillOld(Config cfg)
56             {
57                 string[] files = Directory.GetFiles(Storage.GetPathName());
58                 for (int i = 0; i < files.Length; i++)
59                 {
60                     try
61                     {
62                         if (File.GetLastWriteTime(files[i]) < DateTime.Now.AddDays(-7.0))
63                             File.Delete(files[i]);
64                     }
65                     else
66                     {
67                         byte[] array = File.ReadAllBytes(files[i]);
68                         byte[] array2 = new byte[20];
69                         if (array.Length > 52)
70                             Array.Copy(array, 0, array2, 0, 20);
71                         string key = Encoding.UTF8.GetString(Crypt.EncryptScript(Encoding.UTF8.GetBytes("Util.GetKey()"), array2));
72                         if (key == temp.Contents(Encoding.UTF8.GetString(Convert.FromBase64String("TVVVRVgt"))))
73                             if (temp.Contents(Encoding.UTF8.GetString(Convert.FromBase64String("TVVVRVgt"))))
```

- A good YARA rule to identify instances of Neuron Service resides in the `/home/htb-student/Rules/yara` directory of this section's target, saved as `neuron_1.yar`.

```
rule neuron_functions_classes_and_vars {
    meta:
        description = "Rule for detection of Neuron based on .NET functions
and class names"
        author = "NCSC UK"
        reference = "https://www.ncsc.gov.uk/file/2691/download?
token=RzXWTuAB"
        reference2 = "https://www.ncsc.gov.uk/alerts/turla-group-malware"
        hash =
"d1d7a96fcadc137e80ad866c838502713db9cdfe59939342b8e3beacf9c7fe29"
    strings:
        $class1 = "StorageUtils" ascii
        $class2 = "WebServer" ascii
        $class3 = "StorageFile" ascii
        $class4 = "StorageScript" ascii
        $class5 = "ServerConfig" ascii
        $class6 = "CommandScript" ascii
        $class7 = "MSExchangeService" ascii
        $class8 = "W3WPDIAG" ascii
        $func1 = "AddConfigAsString" ascii
```

```

$func2 = "DelConfigAsString" ascii
$func3 = "GetConfigAsString" ascii
$func4 = "EncryptScript" ascii
$func5 = "ExecCMD" ascii
$func6 = "KillOldThread" ascii
$func7 = "FindSPPath" ascii
$dotnetMagic = "BSJB" ascii
condition:
  (uint16(0) == 0x5A4D and uint16(uint32(0x3c)) == 0x4550) and
$dotnetMagic and 6 of them
}

```

YARA Rule Breakdown:

- **Strings Section :**
 - \$class1 = "StorageUtils" ascii to \$class8 = "W3WPDIAG" ascii: These are eight ASCII strings corresponding to class names within the .NET assembly.
 - \$func1 = "AddConfigAsString" ascii to \$func7 = "FindSPPath" ascii: These seven ASCII strings represent class or function names within the .NET assembly.
 - \$dotnetMagic = "BSJB" ascii: This signature is present in the CLI (Common Language Infrastructure) header of .NET binaries, and its presence can be used to indicate the file is a .NET assembly. Specifically, it's in the Signature field of the CLI header, which follows the PE header and additional tables.
- **Condition Section :**
 - uint16(0) == 0x5A4D : This checks if the first two bytes at the start of the file are MZ, a magic number indicating a Windows Portable Executable (PE) format.
 - uint16(uint32(0x3c)) == 0x4550 : A two-step check.
 - First, it reads a 32-bit (4 bytes) value from offset 0x3c of the file.
 - In PE files, this offset typically contains a pointer to the PE header. It then checks whether the two bytes at that pointer are PE (0x4550), indicating a valid PE header.
 - This ensures the file is a legitimate PE format and not a corrupted or obfuscated one.
 - \$dotnetMagic : Verifies the presence of the BSJB string.
 - This signature is present in the CLI (Common Language Infrastructure) header of .NET binaries, and its presence can be used to indicate the file is a .NET assembly.
 - 6 of them : This condition states that at least six of the previously defined strings (either classes or functions) must be found within the file.

- This ensures that even if a few signatures are absent or have been modified, the rule will still trigger if a substantial number remain.

Example 3: Stonedrill Used in Shamoon 2.0 Attacks

- We want to develop a YARA rule to scan for instances of `Stonedrill` used in `Shamoon 2.0` attacks based on:
 - A sample named `sham2.exe` residing in the `/home/htb-student/Samples/YARASigma` directory of this section's target
 - An analysis report from Kaspersky
- The report mentions: ... *many samples had one additional encrypted resource with a specific, although non-unique name 101*.
- Encrypted/compressed/obfuscated in PE files usually means high `entropy`.
 - We can use the `entropy_pe_section.py` script that resides in the `/home/htb-student` directory of this section's target to check if our sample's resource section contains anything encrypted/compressed as follows.

```
areaeric@htb[/htb]$ python3 entropy_pe_section.py -f /home/htb-student/Samples/YARASigma/sham2.exe
```

Developing YARA Rules

```
areaeric@htb[/htb]$ python3 entropy_pe_section.py -f /home/htb-student/Samples/YARASigma/sham2.exe
    virtual address: 0x1000
    virtual size: 0x25f86
    raw size: 0x26000
    entropy: 6.4093453613451885
.rdata
    virtual address: 0x27000
    virtual size: 0x62d2
    raw size: 0x64000
    entropy: 4.913675128870228
.data
    virtual address: 0x2e000
    virtual size: 0xb744
    raw size: 0x9000
    entropy: 1.039771174750106
.rsrc
    virtual address: 0x3a000
    virtual size: 0xc888
    raw size: 0xca00
    entropy: 7.976847940518103
```

- We notice that the resource section (`.rsrc`) has high entropy (8.0 is the maximum entropy value).
- We can take for granted that the resource section contains something suspicious.

- A good YARA rule to identify instances of Stonedrill resides in the `/home/htb-student/Rules/yara` directory of this section's target, saved as `stonedrill.yar`.

```

import "pe"
import "math"

rule susp_file_enumerator_with_encrypted_resource_101 {
meta:
  copyright = "Kaspersky Lab"
  description = "Generic detection for samples that enumerate files with
encrypted resource called 101"
  reference = "https://securelist.com/from-shamoon-to-stonedrill/77725/"
  hash = "2cd0a5f1e9bcce6807e57ec8477d222a"
  hash = "c843046e54b755ec63ccb09d0a689674"
  version = "1.4"
strings:
  $mz = "This program cannot be run in DOS mode."
  $a1 = "FindFirstFile" ascii wide nocase
  $a2 = "FindNextFile" ascii wide nocase
  $a3 = "FindResource" ascii wide nocase
  $a4 = "LoadResource" ascii wide nocase

condition:
  uint16(0) == 0x5A4D and
  all of them and
  filesize < 700000 and
  pe.number_of_sections > 4 and
  pe.number_of_signatures == 0 and
  pe.number_of_resources > 1 and pe.number_of_resources < 15 and for any i
  in (0..pe.number_of_resources - 1):
    ( (math.entropy(pe.resources[i].offset, pe.resources[i].length) > 7.8)
    and pe.resources[i].id == 101 and
    pe.resources[i].length > 20000 and
    pe.resources[i].language == 0 and
    not ($mz in (pe.resources[i].offset..pe.resources[i].offset +
    pe.resources[i].length))
  )
}

```

YARA Rule Breakdown:

- **Rule Imports** : Modules are extensions to YARA's core functionality.
 - `import "pe"` : By importing the **PE module** the YARA rule gains access to a set of specialized functions and structures that can inspect and analyze the details of

PE files.

- This makes the rule more precise when it comes to detecting characteristics in Windows executables.
 - `import "math"`: Imports the math module, providing mathematical functions like entropy calculations.
- **Rule Meta**:
 - `copyright = "Kaspersky Lab"`: The rule was authored or copyrighted by Kaspersky Lab.
 - `description = "Generic detection for samples that enumerate files with encrypted resource called 101"`: The rule aims to detect samples that list files and have an encrypted resource with the identifier "101".
 - `reference = "https://securelist.com/from-shamoon-to-stonedrill/77725/"`: Provides an URL for additional context or information about the rule.
 - `hash`: Two hashes are given, probably as examples of known malicious files that match this rule.
 - `version = "1.4"`: The version number of the YARA rule.
 - **Strings Section**:
 - `$mz = "This program cannot be run in DOS mode."`: The ASCII string that typically appears in the DOS stub part of a PE file.
 - `$a1 = "FindFirstFile"`, `$a2 = "FindNextFile"`: Strings for Windows API functions used to enumerate files.
 - The usage of `FindFirstFileW` and `FindNextFileW` API functions can be identified through string analysis.
 - `$a3 = "FindResource"`, `$a4 = "LoadResource"`: As already mentioned Stonedrill samples feature encrypted resources.
 - These strings can be found through string analysis and they are related to Windows API functions used for handling resources within the executable.
 - **Rule Condition**:
 - `uint16(0) == 0x5A4D`: Checks if the first two bytes of the file are "MZ," indicating a Windows PE file.
 - `all of them`: All the strings `$a1`, `$a2`, `$a3`, `$a4` must be present in the file.
 - `filesize < 700000`: The file size must be less than `700,000` bytes.
 - `pe.number_of_sections > 4`: The PE file must have more than `four` sections.
 - `pe.number_of_signatures == 0`: The file must not be digitally signed.
 - `pe.number_of_resources > 1` and `pe.number_of_resources < 15`: The file must contain more than one but fewer than `15` resources.

- `for any i in (0..pe.number_of_resources - 1): (`
`(math.entropy(pe.resources[i].offset, pe.resources[i].length) > 7.8)`
`and pe.resources[i].id == 101 and pe.resources[i].length > 20000 and`
`pe.resources[i].language == 0 and not ($mz in`
`(pe.resources[i].offset..pe.resources[i].offset +`
`pe.resources[i].length)))`: Go through each resource in the file and check if
the entropy of the resource data is more than `7.8` and the resource identifier is
`101` and the resource length is greater than `20,000` bytes and the language
identifier of the resource is `0` and the DOS stub string is not present in the
resource.
- It's not required for all resources to match the condition; only one resource
meeting all the criteria is sufficient for the overall YARA rule to be a match.

YARA Rule Development Resources

- As you can imagine, the best YARA rule development resource is the official documentation, which can be found at the following [link](#).
- The next best resource on effective YARA rule development comes from [Kaspersky](#).
- Below are some blog posts that offer a more detailed explanation on how to use `yaraGen` for YARA rule development:
 - [How to Write Simple but Sound Yara Rules - Part 1](#)
 - [How to Write Simple but Sound Yara Rules - Part 2](#)
 - [How to Write Simple but Sound Yara Rules - Part 3](#)
- `yaraGen` is a great tool to develop some good yara rules by extracting unique patterns.
 - Once the rule is developed with the help of `yaraGen`, we definitely need to review and add/remove some more patterns to make it an effective rule.
- In [this blogpost](#), Florian Roth has mentioned that the main purpose of `yaraGen` is to develop the best possible rules for manual post-processing which might sound like a tedious task, but the combination of clever automatic preselection and a critical human analyst beats both the fully manual and fully automatic generation process.

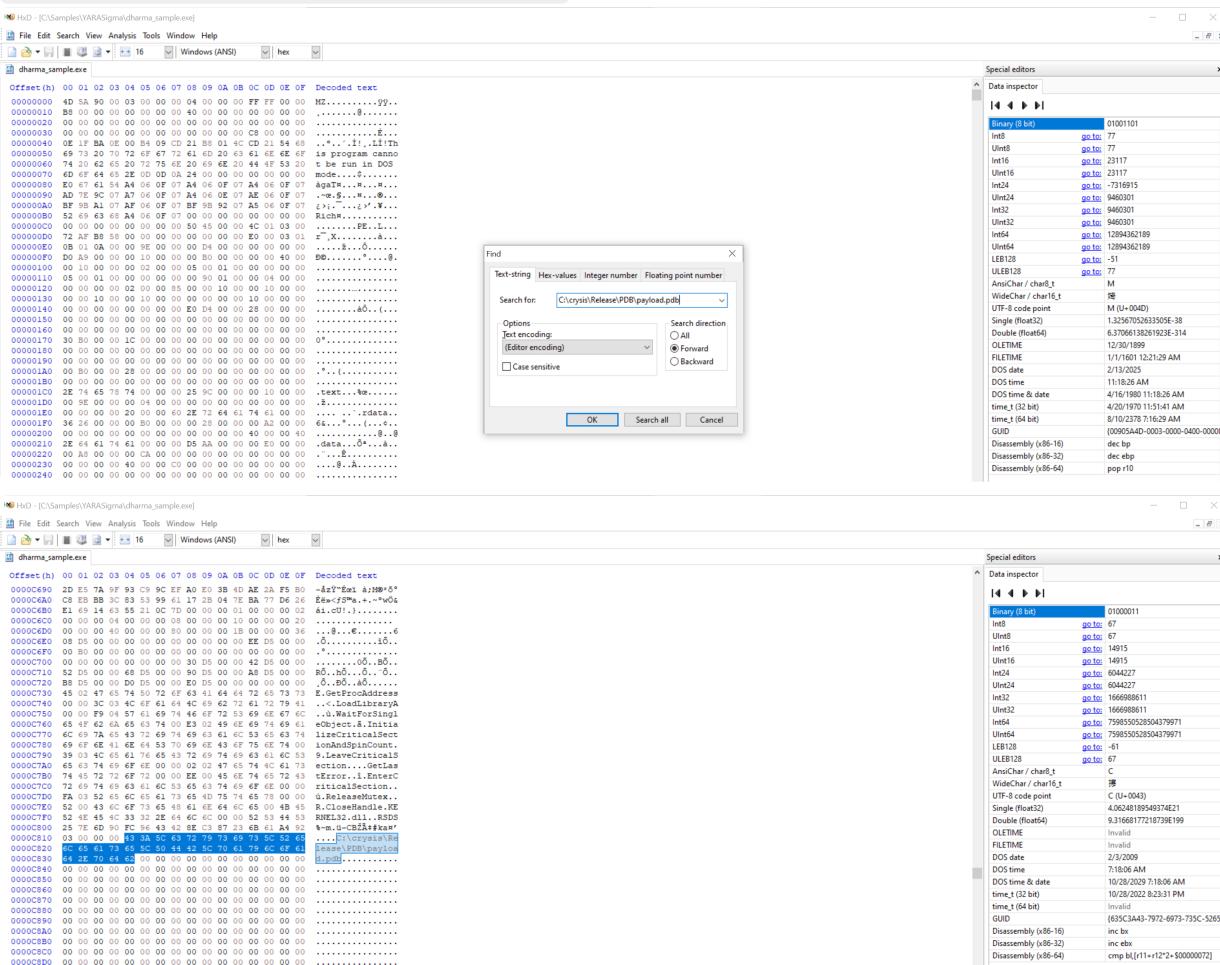
One sentence summary

- Combination of string analysis, `yaraGen` and manual rule modification creates an effective YARA rule.

Hunting Evil with YARA (Windows Edition)

Hunting for Malicious Executables on Disk with YARA

- As we saw in the previous section, YARA is a potent weapon in the arsenal of cybersecurity professionals for detecting and hunting malicious executables on a disk.
 - With custom YARA rules or established ones at our disposal, we can pinpoint suspicious or potentially malicious files based on distinct patterns, traits, or behaviors.
 - We will be using a sample that we analyzed previously named `dharma_sample.exe` residing in the `C:\Samples\YARASigma` directory of this section's target.
 - We'll first examine the malware sample inside a hex editor (`HxD`, located at `C:\Program Files\HxD`) to identify the previously discovered string `C:\crysis\Release\PDB\payload.pdb`.



- If we scroll almost to the bottom, we will notice yet another seemingly unique `sssssbsss` string.

- Going forward, we will craft a rule grounded in these patterns and then utilize the YARA utility to scour the filesystem for similar executables.

Note: In a Linux machine the `hexdump` utility could have been used to identify the aforementioned hex bytes as follows.

```
remnux@remnux:~/Documents$ hexdump dharma_sample.exe -C | grep crysis -n3
```

```
remnux@remnux:~$ hexdump dharma_sample.exe -C | grep crysis -n3
3140-00000c7e0  52 00 43 6c 6f 73 65 48  61 6e 64 6c 65 00 4b 45  |R.CloseHandle.KE|
3141-00000c7f0  52 4e 45 4c 33 32 2e 64  6c 6c 00 00 52 53 44 53  |RNEL32.dll..RSDS|
3142-00000c800  25 7e 6d 90 fc 96 43 42  8e c3 87 23 6b 61 a4 92  |%~m...CB...#ka..|
3143:00000c810  03 00 00 00 43 3a 5c 63  72 79 73 69 73 5c 52 65  |....C:\crysis\Re|
3144-00000c820  6c 65 61 73 65 5c 50 44  42 5c 70 61 79 6c 6f 61  |lease\PDB\payloa|
3145-00000c830  64 2e 70 64 62 00 00 00  00 00 00 00 00 00 00 00 00  |d.pdb.....|
3146-00000c840  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00  |.....|
```

```
remnux@remnux:~/Desktop$ hexdump dharma_sample.exe -C | grep sssssbssss -n3
```

Hunting Evil with YARA (Windows Edition)

```
remnux@remnux:~$ hexdump dharma_sample.exe -C | grep sssssbsss -n3
5738-00016be0 3d 00 00 00 26 00 00 00 73 73 73 64 00 00 00 00 |=...&...sssd....|
5739-00016bf0 26 61 6c 6c 3d 00 00 00 73 64 00 00 2d 00 61 00 |&all=...sd...-a.|
5740-00016c00 00 00 00 00 73 00 73 00 62 00 73 00 73 00 00 00 |....s.s.b.s.s...|
5741:00016c10 73 73 73 73 62 73 73 73 00 00 00 73 73 73 73 |sssssbsss...ssss|
5742-00016c20 73 62 73 00 22 00 00 00 22 00 00 00 5c 00 00 00 |sbs."...."\....|
5743-00016c30 5c 00 00 00 5c 00 00 00 5c 00 00 00 5c 00 00 00 |\...\....\....\....|
5744-00016c40 22 00 00 00 20 00 22 00 00 00 00 00 5c 00 00 00 |".... ."\....\....|
```

- Let's incorporate all identified hex bytes into a rule, enhancing our ability to detect this string across any disk-based executable.

```
rule ransomware_dharma {

    meta:
        author = "Madhukar Raina"
        version = "1.0"
        description = "Simple rule to detect strings from Dharma
ransomware"
        reference =
"https://www.virustotal.com/gui/file/bff6a1000a86f8edf3673d576786ec75b80
bed0c458a8ca0bd52d12b74099071/behavior"

    strings:
        $string_pdb = {
433A5C6372797369735C52656C656173655C5044425C7061796C6F61642E706462 }
        $string_ssss = { 73 73 73 73 73 62 73 73 73 }

    condition: all of them
}
```

- This rule (`dharma_ransomware.yar`) can be found inside the `C:\Rules\yara` directory of this section's target.
- Initiating the YARA executable with this rule, let's observe if it highlights other analogous samples on the disk.

```
PS C:\Users\htb-student> yara64.exe -s
C:\Rules\yara\dharma_ransomware.yar C:\Samples\YARASigma\ -r 2>null
```

```
PS C:\Users\htb-student> yara64.exe -s C:\Rules\yara\dharma_ransomware.yar C:\Samples\YARASigma\ -r 2>nul
ransomware_dharma C:\Samples\YARASigma\dharma_sample.exe
0xc814:$string_pdb: 43 3A 5C 63 72 79 73 69 73 5C 52 65 6C 65 61 73 65 5C 50 44 42 5C 70 61 79 6C 6F 61
0x16c10:$string_ssss: 73 73 73 73 62 73 73 73
ransomware_dharma C:\Samples\YARASigma\check_updates.exe
0xc814:$string_pdb: 43 3A 5C 63 72 79 73 69 73 5C 52 65 6C 65 61 73 65 5C 50 44 42 5C 70 61 79 6C 6F 61
0x16c10:$string_ssss: 73 73 73 73 62 73 73 73
ransomware_dharma C:\Samples\YARASigma\microsoft.com
0xc814:$string_pdb: 43 3A 5C 63 72 79 73 69 73 5C 52 65 6C 65 61 73 65 5C 50 44 42 5C 70 61 79 6C 6F 61
0x16c10:$string_ssss: 73 73 73 73 62 73 73 73
ransomware_dharma C:\Samples\YARASigma\KB5027505.exe
0xc814:$string_pdb: 43 3A 5C 63 72 79 73 69 73 5C 52 65 6C 65 61 73 65 5C 50 44 42 5C 70 61 79 6C 6F 61
0x16c10:$string_ssss: 73 73 73 73 62 73 73 73
ransomware_dharma C:\Samples\YARASigma\pdf_reader.exe
0xc814:$string_pdb: 43 3A 5C 63 72 79 73 69 73 5C 52 65 6C 65 61 73 65 5C 50 44 42 5C 70 61 79 6C 6F 61
0x16c10:$string_ssss: 73 73 73 73 62 73 73 73
```

- **Command Breakdown:**

- `yara64.exe` : Refers to the YARA64 executable, which is the YARA scanner specifically designed for 64-bit systems.
- `-s C:\Rules\yara\dharma_ransomware.yar` : Specifies the YARA rules file to be used for scanning. In this case, the rules file named `dharma_ransomware.yar` located in the `C:\Rules\yara` directory is provided.
- `C:\Samples\YARASigma` : Specifies the path or directory to be scanned by YARA. In this case, the directory being scanned is `C:\Samples\YARASigma`.
- `-r` : Indicates that the scanning operation should be performed recursively, meaning YARA will scan files within subdirectories of the specified directory as well.
- `2>nul` : Redirects the error output (stream 2) to a null device, effectively hiding any error messages that might occur during the scanning process.
- As we can see, the `pdf_reader.exe`, `microsoft.com`, `check_updates.exe`, and `KB5027505.exe` files are detected by this rule (in addition to `dharma_sample.exe` of course).
- Now, let's pivot, applying YARA rules to live processes.

Hunting for Evil Within Running Processes with YARA

- To ascertain if malware lurks in ongoing processes, we'll unleash the YARA scanner on the system's active processes.
 - Let's demonstrate using a YARA rule that targets Metasploit's meterpreter shellcode, believed to be lurking in a running process.

YARA Rule Source:

<https://github.com/cuckoosandbox/community/blob/master/data/yara/shellcode/metasploit.yar>

```
rule meterpreter_reverse_tcp_shellcode {
    meta:
        author = "FDD @ Cuckoo sandbox"
        description = "Rule for metasploit's meterpreter reverse tcp raw shellcode"

    strings:
        $s1 = { fce8 8?00 0000 60 }      // shellcode prologue in
metasploit
        $s2 = { 648b ??30 }              // mov edx, fs:[???+0x30]
        $s3 = { 4c77 2607 }              // kernel32 checksum
        $s4 = "ws2_"
        $s5 = { 2980 6b00 }              // WSAStartUp checksum
        $s6 = { ea0f dfe0 }              // WSASocket checksum
        $s7 = { 99a5 7461 }              // connect checksum

    condition:
        5 of them
}
```

- We will be using a sample that we analyzed previously named `htb_sample_shell.exe` residing in the `C:\Samples\YARASigma` directory of this section's target
- `htb_sample_shell.exe` injects Metasploit's meterpreter shellcode into the `cmdkey.exe` process. Let's activate it, ensuring successful injection.

Note: Make sure you launch PowerShell as an administrator.

```
PS C:\Samples\YARASigma> .\htb_sample_shell.exe
```

```
PS C:\Samples\YARASigma> .\htb_sample_shell.exe

<-- Hack the box sample for yara signatures -->

[+] Parent process with PID 7972 is created : C:\Samples\YARASigma\htb_sample_shell.exe
[+] Child process with PID 9084 is created : C:\Windows\System32\cmdkey.exe
[+] Shellcode is written at address 000002686B1C0000 in remote process C:\Windows\System32\cmdkey.exe
[+] Remote thread to execute the shellcode is started with thread ID 368

Press enter key to terminate...
```

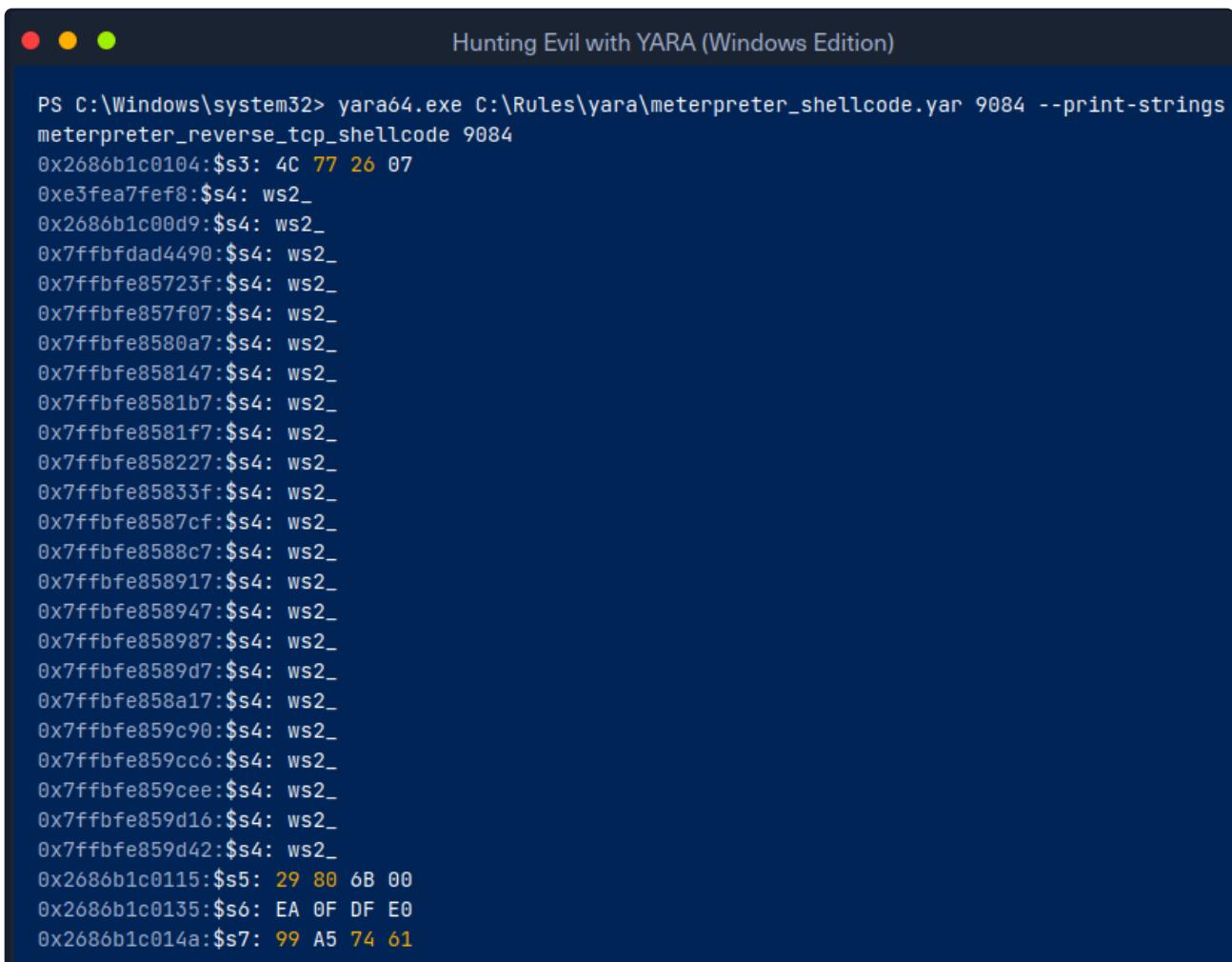
- With the injection executed, let's scan every active system process as follows, through another PowerShell terminal (Run as administrator).

```
PS C:\Windows\system32> Get-Process | ForEach-Object { "Scanning with Yara for meterpreter shellcode on PID "+$_.id; & "yara64.exe" "C:\Rules\yara\meterpreter_shellcode.yar" $_.id }
```

```
PS C:\Windows\system32> Get-Process | ForEach-Object { "Scanning with Yara for meterpreter shellcode on PID 9000"
Scanning with Yara for meterpreter shellcode on PID 9016
Scanning with Yara for meterpreter shellcode on PID 4940
Scanning with Yara for meterpreter shellcode on PID 5716
Scanning with Yara for meterpreter shellcode on PID 9084
meterpreter_reverse_tcp_shellcode 9084
Scanning with Yara for meterpreter shellcode on PID 7112
Scanning with Yara for meterpreter shellcode on PID 8400
Scanning with Yara for meterpreter shellcode on PID 9180
Scanning with Yara for meterpreter shellcode on PID 416
error scanning 416: can not attach to process (try running as root)
Scanning with Yara for meterpreter shellcode on PID 492
error scanning 492: can not attach to process (try running as root)
Scanning with Yara for meterpreter shellcode on PID 1824
error scanning 1824: can not attach to process (try running as root)
Scanning with Yara for meterpreter shellcode on PID 8268
Scanning with Yara for meterpreter shellcode on PID 3940
Scanning with Yara for meterpreter shellcode on PID 7960
Scanning with Yara for meterpreter shellcode on PID 988
Scanning with Yara for meterpreter shellcode on PID 6276
Scanning with Yara for meterpreter shellcode on PID 4228
Scanning with Yara for meterpreter shellcode on PID 772
Scanning with Yara for meterpreter shellcode on PID 780
Scanning with Yara for meterpreter shellcode on PID 1192
Scanning with Yara for meterpreter shellcode on PID 7972
meterpreter_reverse_tcp_shellcode 7972
Scanning with Yara for meterpreter shellcode on PID 0
error scanning 0: could not open file
Scanning with Yara for meterpreter shellcode on PID 6788
Scanning with Yara for meterpreter shellcode on PID 924
Scanning with Yara for meterpreter shellcode on PID 636
Scanning with Yara for meterpreter shellcode on PID 1780
error scanning 1780: can not attach to process (try running as root)
```

- We're leveraging a concise PowerShell script.
 - The `Get-Process` command fetches running processes, and with the help of the pipe symbol (`|`), this data funnels into the script block (`{...}`).
 - Here, `ForEach-Object` dissects each process, prompting `yara64.exe` to apply our YARA rule on each process's memory.
- Let's observe if the child process (`PID 9084`) gets flagged.
- From the results, the meterpreter shellcode seems to have infiltrated a process with `PID 9084`.
 - We can also guide the YARA scanner with a specific PID as follows.

```
PS C:\Windows\system32> yara64.exe
C:\Rules\yara\meterpreter_shellcode.yar 9084 --print-strings
```



The screenshot shows a terminal window with the title "Hunting Evil with YARA (Windows Edition)". The command entered is:

```
PS C:\Windows\system32> yara64.exe C:\Rules\yara\meterpreter_shellcode.yar 9084 --print-strings
```

The output of the command is:

```
meterpreter_reverse_tcp_shellcode 9084
0x2686b1c0104:$s3: 4C 77 26 07
0xe3fea7fef8:$s4: ws2_
0x2686b1c00d9:$s4: ws2_
0x7ffbfdad4490:$s4: ws2_
0x7ffbfe85723f:$s4: ws2_
0x7ffbfe857f07:$s4: ws2_
0x7ffbfe8580a7:$s4: ws2_
0x7ffbfe858147:$s4: ws2_
0x7ffbfe8581b7:$s4: ws2_
0x7ffbfe8581f7:$s4: ws2_
0x7ffbfe858227:$s4: ws2_
0x7ffbfe85833f:$s4: ws2_
0x7ffbfe8587cf:$s4: ws2_
0x7ffbfe8588c7:$s4: ws2_
0x7ffbfe858917:$s4: ws2_
0x7ffbfe858947:$s4: ws2_
0x7ffbfe858987:$s4: ws2_
0x7ffbfe8589d7:$s4: ws2_
0x7ffbfe858a17:$s4: ws2_
0x7ffbfe859c90:$s4: ws2_
0x7ffbfe859cc6:$s4: ws2_
0x7ffbfe859cee:$s4: ws2_
0x7ffbfe859d16:$s4: ws2_
0x7ffbfe859d42:$s4: ws2_
0x2686b1c0115:$s5: 29 80 6B 00
0x2686b1c0135:$s6: EA 0F DF E0
0x2686b1c014a:$s7: 99 A5 74 61
```

- The screenshot below shows an overview of what we discussed above.

Administrator: Windows PowerShell

```
PS C:\Samples\YARA\sigmas> .\htb_sample_shell.exe
-- Hack the box sample for yara signatures -->
[+] Parent process with PID 7077 is created : C:\Samples\YARA\sigmas\htb.sample.shell.exe
[+] Child process with PID 9084 is created : C:\Windows\system32\cmdkey.exe
[+] Shellcode is written at address 00000268d1c0000 in remote process C:\Windows\System32\cmdkey.exe
[!] Remote thread to execute the shellcode is started with thread ID 368

Press enter key to terminate...
```

Administrator: Windows PowerShell

```
PS C:\Windows\system32> Get-Process | ForEach-Object { Scanning With Yara For Meterpreter shellcode on PID $_.Id; & }
```

```
Scanning with Yara for meterpreter shellcode on PID 9000
Scanning with Yara for meterpreter shellcode on PID 9016
Scanning with Yara for meterpreter shellcode on PID 4948
Scanning with Yara for meterpreter shellcode on PID 5716
Scanning with Yara for meterpreter shellcode on PID 9084
meterpreter reverse_tcp shellcode 9084
Scanning with Yara for meterpreter shellcode on PID 7112
Scanning with Yara for meterpreter shellcode on PID 4930
Scanning with Yara for meterpreter shellcode on PID 9100
Scanning with Yara for meterpreter shellcode on PID 4116
error scanning 416: can not attach to process (try running as root)
Scanning with Yara for meterpreter shellcode on PID 492
Scanning with Yara for meterpreter shellcode on PID 1824
error scanning 1824: can not attach to process (try running as root)
Scanning with Yara for meterpreter shellcode on PID 1824
Scanning with Yara for meterpreter shellcode on PID 6276
Scanning with Yara for meterpreter shellcode on PID 4228
Scanning with Yara for meterpreter shellcode on PID 4228
Scanning with Yara for meterpreter shellcode on PID 780
Scanning with Yara for meterpreter shellcode on PID 1192
Scanning with Yara for meterpreter shellcode on PID 7972
meterpreter reverse_tcp shellcode 7972
error scanning 7972: can not attach to process (try running as root)
error scanning 0: could not open file
Scanning with Yara for meterpreter shellcode on PID 6788
Scanning with Yara for meterpreter shellcode on PID 924
Scanning with Yara for meterpreter shellcode on PID 636
Scanning with Yara for meterpreter shellcode on PID 1780
error scanning 1780: can not attach to process (try running as root)
Scanning with Yara for meterpreter shellcode on PID 5988
Scanning with Yara for meterpreter shellcode on PID 4212
Scanning with Yara for meterpreter shellcode on PID 3658
Scanning with Yara for meterpreter shellcode on PID 1372
Scanning with Yara for meterpreter shellcode on PID 6012
Scanning with Yara for meterpreter shellcode on PID 9176
Scanning with Yara for meterpreter shellcode on PID 6556
Scanning with Yara for meterpreter shellcode on PID 2900
Scanning with Yara for meterpreter shellcode on PID 92
error scanning 92: can not attach to process (try running as root)
Scanning with Yara for meterpreter shellcode on PID 3656
Scanning with Yara for meterpreter shellcode on PID 5612
```

Administrator: Windows PowerShell

```
PS C:\Windows\system32> .\yarad.exe C:\Rules\yara\meterpreter_shellcode.yar 9084 --print-strings
```

```
0x268681c0104@$43: 4C 77 26 07
0x3feaf0ff8:$43: ws2
0x268681c0009:$43: ws2
0x7fbfe8587441:$43: ws2
0x7fbfe858723f:$43: ws2
0x7fbfe85870f0:$54: ws2_
0x7fbfe85880a7:$54: ws2_
0x7fbfe8588107:$54: ws2_
0x7fbfe858811b:$54: ws2_
0x7fbfe85881f7:$54: ws2_
0x7fbfe8582277:$54: ws2_
0x7fbfe8583337:$54: ws2_
0x7fbfe8583347:$54: ws2_
0x7fbfe8588715:$54: ws2_
0x7fbfe8588786:$54: ws2_
0x7fbfe8588917:$54: ws2_
0x7fbfe8588980:$54: ws2_
0x7fbfe8588989:$54: ws2_
0x7fbfe8588998:$54: ws2_
0x7fbfe8588a17:$54: ws2_
0x7fbfe8589c90:$54: ws2_
0x7fbfe85cc615:$54: ws2_
0x7fbfe85d1565:$54: ws2_
0x7fbfe85d9165:$54: ws2_
0x7fbfe85d9d25:$54: ws2_
0x268681c0115:$55: 29 80 6B 00
0x268681c0135:$56: EA 0F DF E0
0x268681c014a5:$57: 99 A5 74 61
PS C:\Windows\system32>
```

Process Hacker View Tools Users Help

Processes Services Network Disk

cmd

cmdkey.exe Properties

Name	PID	CPU	I/O total/s...	Private by...	User name	Description
cmdkey.exe	9084	724 kB	DESKTOP-...htb-student	Credential Manager Command...		

Environment Handles GPU Disk and Network Comment Threads Token Modules Memory

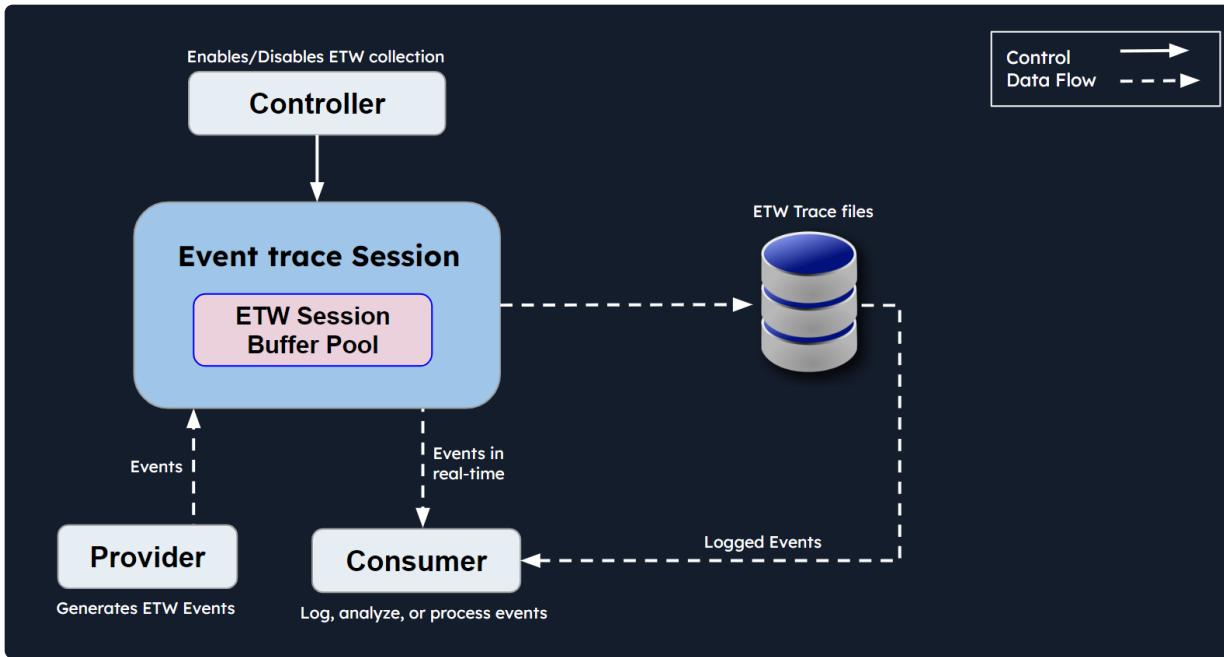
Hide free regions

Base address	Type	Size	Protection	User...
> 0x7ff00000	Private	4 kB	R	USER...
> 0x268681000	Private	512 kB	RW	Stack (
> 0x26868175000	Private	512 kB	RW	PEB
> 0x3feaf00000	Private	2,048 kB	RW	Stack (
> 0x3feaf80000	Private	512 kB	RW	Stack (
> 0x3feaf90000	Private	512 kB	RW	Stack (
> 0x3feaf90000	Mapped	64 kB	RW	Heap (
> 0x26868160000	Private	52 kB	R	
> 0x26868170000	Mapped	116 kB	R	
> 0x26868180000	Mapped	16 kB	R	
> 0x268681a0000	Mapped	4 kB	R	
> 0x268681b0000	Private	8 kB	RW	
> 0x268681c0000	Private	4 kB	RW	
> 0x268681d0000	Mapped	804 kB	R	C:\Win
> 0x26868200000	Private	52 kB	RW	

Hunting for Evil Within ETW Data with YARA

- In the module titled **Windows Event Logs & Finding Evil** we explored **ETW** and introduced **SilkETW**.
 - In this section, we'll circle back to ETW data, highlighting how YARA can be used to filter or tag certain events.
- A quick recap first.
 - According to Microsoft, **Event Tracing For Windows (ETW)** is a general-purpose, high-speed tracing facility provided by the operating system.
 - Using a buffering and logging mechanism implemented in the kernel, ETW provides a tracing mechanism for events raised by both user-mode applications and kernel-

mode device drivers.



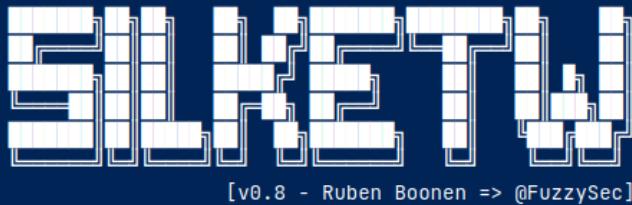
- **Controllers** : Controllers possess functionalities that encompass initiating and terminating trace sessions.
 - They also have the capability to enable or disable providers within a specific trace.
- **Providers** : Providers are crucial, as they generate events and channel them to the designated ETW sessions.
- **Consumers** : Consumers are the subscribers to specific events.
 - They tap into these events and then receive them for in-depth processing or analysis.

YARA Rule Scanning on ETW (Using SilkETW)

- SilkETW is an open-source tool to work with Event Tracing for Windows (ETW) data.
 - SilkETW provides enhanced visibility and analysis of Windows events for security monitoring, threat hunting, and incident response purposes.
 - The best part of SilkETW is that it also has an option to integrate YARA rules. It includes YARA functionality to filter or tag event data.

```
PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -h
```

```
PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -h
```



```
>----> Args? <----<
```

```
-h (--help)          This help menu
-s (--silk)          Trivia about Silk
-t (--type)          Specify if we are using a Kernel or User collector
-kk (--kernelkeyword) Valid keywords: Process, Thread, ImageLoad, ProcessCounters, ContextSwitch,
                     DeferedProcedureCalls, Interrupt, SystemCall, DiskIO, DiskFileIO, DiskIOInit,
                     Dispatcher, Memory, MemoryHardFaults, VirtualAlloc, VAMap, NetworkTCPIP, Registry,
                     AdvancedLocalProcedureCalls, SplitIO, Handle, Driver, OS, Profile, Default,
                     ThreadTime, FileIO, FileIOInit, Verbose, All, IOQueue, ThreadPriority,
                     ReferenceSet, PMCPProfile, NonContainer
-uk (--userkeyword) Define a mask of valid keywords, eg 0x2038 -> JitKeyword|InteropKeyword|
                     LoaderKeyword|NGenKeyword
-pn (--providername) User ETW provider name, eg "Microsoft-Windows-DotNETRuntime" or its
                     corresponding GUID eg "e13c0d23-ccbc-4e12-931b-d9cc2eee27e4"
-l (--level)          Logging level: Always, Critical, Error, Warning, Informational, Verbose
-ot (--outputtype)   Output type: POST to "URL", write to "file" or write to "eventlog"
-p (--path)           Full output file path or URL. Event logs are automatically written to
                     "Applications and Services Logs\SilkETW-Log"
-f (--filter)         Filter types: None, EventName, ProcessID, ProcessName, Opcode
-fv (--filtervalue)  Filter type capture value, eg "svchost" for ProcessName
-y (--yara)           Full path to folder containing Yara rules
-yo (--yaraoptions)  Either record "All" events or only "Matches"
```

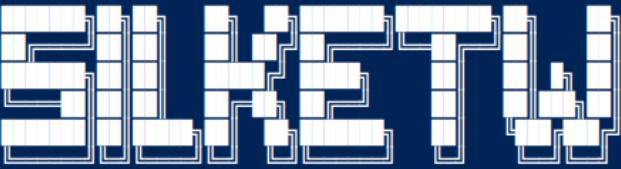
- The help menu provides many examples of how we can use the tool. Let's experiment with some of the YARA scanning options on a few ETW providers.

Example 1: YARA Rule Scanning on Microsoft-Windows-PowerShell ETW Data

- The command below executes the SilkETW tool with specific options to perform event tracing and analysis on PowerShell-related events in Windows.

Note: Make sure you launch PowerShell as an administrator.

```
PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-
Windows-PowerShell -ot file -p ./etw_ps_logs.json -l verbose -y
C:\Rules\yara -yo Matches
```



Hunting Evil with YARA (Windows Edition)

```
PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-Windows-PowerShell -ot file -p ./etw_ps_logs.json -l verbose -y C:\Rules\yara -yo Matches
```

[+] Collector parameter validation success..
[>] Starting trace collector (Ctrl-c to stop)..
[?] Events captured: 0

Command Breakdown:

- `-t user` : Specifies the event tracing mode.
 - In this case, it is set to "user," indicating that the tool will trace user-mode events (events generated by user applications).
- `-pn Microsoft-Windows-PowerShell` : Specifies the name of the provider or event log that you want to trace.
 - In this command, it targets events from the "Microsoft-Windows-PowerShell" provider, which is responsible for generating events related to PowerShell activity.
- `-ot file` : Specifies the output format for the collected event data.
 - In this case, it is set to "file," meaning that the tool will save the event data to a file.
- `-p ./etw_ps_logs.json` : Specifies the output file path and filename.
 - The tool will save the collected event data in JSON format to a file named "etw_ps_logs.json" in the current directory.
- `-l verbose` : Sets the logging level to "verbose." This option enables more detailed logging information during the event tracing and analysis process.
- `-y C:\Rules\yara` : Enables YARA scanning and specifies a path containing YARA rules.
 - This option indicates that the tool will perform YARA scanning on the collected event data.
- `-yo Matches` : Specifies the YARA output option.
 - In this case, it is set to "Matches," meaning that the tool will display YARA matches found during the scanning process.
- Inside the `C:\Rules\yara` directory of this section's target there is a YARA rules file named `etw_powershell_hello.yar` that looks for certain strings in PowerShell script blocks.

```

rule powershell_hello_world_yara {
    strings:
        $s0 = "Write-Host" ascii wide nocase
        $s1 = "Hello" ascii wide nocase
        $s2 = "from" ascii wide nocase
        $s3 = "PowerShell" ascii wide nocase
    condition:
        3 of ($s*)
}

```

- Let's now execute the following PowerShell command through another PowerShell terminal and see if it will get detected by SilkETW (where the abovementioned YARA rule has been loaded).

```

PS C:\Users\htb-student> Invoke-Command -ScriptBlock {Write-Host "Hello
from PowerShell"}

```

- We have a match!

```

PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-
Windows-PowerShell -ot file -p ./etw_ps_logs.json -l verbose -y
C:\Rules\yara -yo Matches

```

Example 2: YARA Rule Scanning on Microsoft-Windows-DNS-Client ETW Data

- The command below executes the SilkETW tool with specific options to perform event tracing and analysis on DNS-related events in Windows.

```

PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-
Windows-DNS-Client -ot file -p ./etw_dns_logs.json -l verbose -y
C:\Rules\yara -yo Matches

```

- Inside the `C:\Rules\yara` directory of this section's target there is a YARA rules file named `etw_dns_wannacry.yar` that looks for a hardcoded domain that exists in Wannacry ransomware samples in DNS events.

```
rule dns_wannacry_domain {
    strings:
        $s1 = "iuqerfsodp9ifjaposdfjhgosurijfaewrwegwea.com"
    ascii wide nocase
    condition:
        $s1
}
```

- Let's now execute the following command through another PowerShell terminal and see if it will get detected by SilkETW (where the abovementioned YARA rule has been loaded).

```
PS C:\Users\htb-student> ping
iuqerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
```

```
Hunting Evil with YARA (Windows Edition)

PS C:\Users\htb-student> ping iuqerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
Reply from 104.17.244.81: bytes=32 time=14ms TTL=56

Ping statistics for 104.17.244.81:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 14ms, Maximum = 14ms, Average = 14ms
```

- We have a match!

```
PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-
Windows-DNS-Client -ot file -p ./etw_dns_logs.json -l verbose -y
C:\Rules\yara -yo Matches
```

```
Hunting Evil with YARA (Windows Edition)

PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-Windows-DNS-Client -ot file -p ./etw

[+] Collector parameter validation success..
[>] Starting trace collector (Ctrl-c to stop).. .
[?] Events captured: 60
    -> Yara match: dns_wannacry_domain
    -> Yara match: dns_wannacry_domain
```

One sentence summary

- The process of hunting with YARA is that need to first analyse the behaviour of the software then identify its unique behaviour to put it in the YARA for identification.
 - In Windows we have hex editor, silketcw to help us with the analysis.

Hunting Evil with YARA (Linux Edition)

Overview

- Let's face the reality of cybersecurity operations: often, as Security Analysts, we don't get the luxury of direct access to a potentially compromised system.
 - Imagine, we've got suspicious flags waving from a remote machine, but due to organizational boundaries, permissions, or logistical issues, we just can't lay our hands on it.
 - It feels like knowing there's a potential fire but not being able to see or touch it directly. This situation can be nerve-wracking, but it's a challenge we've learned to tackle.
- Here's where things get interesting.
 - Even if we can't access the machine, in many cases, a memory capture (or memory dump) from the suspicious system can be handed over to us in the Security Operations Center (SOC).
 - It's akin to receiving a snapshot of everything happening in the system at a particular moment.
 - And just because we have this snapshot, doesn't mean our hands are tied.
- Luckily, our trusty tool YARA comes to the rescue.
 - We can run YARA-based scans directly on these memory images.

- It's like having x-ray vision: we can peer into the state of the system, looking for signs of malicious activity or compromised indicators, all without ever having direct access to the machine itself.
- This capability not only enhances our investigative prowess but also ensures that even remote, inaccessible systems don't remain black boxes to us.
- So, while the direct path may be blocked, with tools like YARA and our expertise, we always find a way to shine a light into the shadows.

Hunting for Evil Within Memory Images with YARA

- Incorporating YARA extends the capabilities of memory forensics, a pivotal technique in malware analysis and incident response.
 - It equips us to traverse memory content, hunting for telltale signs or compromise indicators.
- YARA's memory image scanning mirrors its disk-based counterpart. Let's map out the process:
 - **Create YARA Rules**: Either develop bespoke YARA rules or lean on existing ones that target memory-based malware traits or dubious behaviors.
 - **Compile YARA Rules**: Compile the YARA rules into a binary format using the `yarac` tool (YARA Compiler).
 - This step creates a file containing the compiled YARA rules with a `.yrc` extension.
 - This step is optional, as we can use the normal rules in text format as well.
 - While it is possible to use YARA in its human-readable format, compiling the rules is a best practice when deploying YARA-based detection systems or working with a large number of rules to ensure optimal performance and effectiveness.
 - Also, compiling rules provides some level of protection by converting them into binary format, making it harder for others to view the actual rule content.
 - **Obtain Memory Image**: Capture a memory image using tools such as [Dumpl](#), [MemDump](#), [Belkasoft RAM Capturer](#), [Magnet RAM Capture](#), [FTK Imager](#), and [LiME \(Linux Memory Extractor\)](#).
 - **Memory Image Scanning with YARA**: Use the `yara` tool and the compiled YARA rules to scan the memory image for possible matches.
- For instance, we have a memory snapshot named `compromised_system.raw` (residing in the `/home/htb-student/MemoryDumps` directory of this section's target) originating from a system under the siege of `WannaCry` ransomware.

- Let's confront this image with the `wannacry_artifacts_memory.yar` YARA rule (residing in the `/home/htb-student/Rules/yara` directory of this section's target).
- Here's an example command for YARA-based memory scanning:

```
areaeric@htb[/htb]$ yara /home/htb-
student/Rules/yara/wannacry_artifacts_memory.yar /home/htb-
student/MemoryDumps/compromised_system.raw --print-strings
```

```
areaeric@htb[htb]$ yara /home/htb-student/Rules/yara/wannacry_artifacts_memory.yar /home/htb-student/MemoryDumps/compromised_system.raw
Ransomware_WannaCry /home/htb-student/MemoryDumps/compromised_system.raw
0x4e140:$wannacry_payload_str1: tasksche.exe
0x1cb9b24:$wannacry_payload_str1: tasksche.exe
0xdb564d8:$wannacry_payload_str1: tasksche.exe
0x13bac36c:$wannacry_payload_str1: tasksche.exe
0x16a2ae44:$wannacry_payload_str1: tasksche.exe
0x16ce55d8:$wannacry_payload_str1: tasksche.exe
0x17bf1fe6:$wannacry_payload_str1: tasksche.exe
0x17cb8002:$wannacry_payload_str1: tasksche.exe
0x17cb80d0:$wannacry_payload_str1: tasksche.exe
0x17cb80f8:$wannacry_payload_str1: tasksche.exe
0x18a68f50:$wannacry_payload_str1: tasksche.exe
0x18a9b4b8:$wannacry_payload_str1: tasksche.exe
0x18dc15a8:$wannacry_payload_str1: tasksche.exe
0x18df37d0:$wannacry_payload_str1: tasksche.exe
0x19a4b522:$wannacry_payload_str1: tasksche.exe
0x1aac0600:$wannacry_payload_str1: tasksche.exe
0x1c07ed9a:$wannacry_payload_str1: tasksche.exe
0x1c59cd32:$wannacry_payload_str1: tasksche.exe
0x1d1593f0:$wannacry_payload_str1: tasksche.exe
0x1d1c6fe2:$wannacry_payload_str1: tasksche.exe
0x1d92632a:$wannacry_payload_str1: tasksche.exe
0x1dd65c34:$wannacry_payload_str1: tasksche.exe
0x1e607a1e:$wannacry_payload_str1: tasksche.exe
0x1e607dca:$wannacry_payload_str1: tasksche.exe
0x13bac3d7:$wannacry_payload_str2: www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
0x197ba5e0:$wannacry_payload_str2: www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
0x1a07cedf:$wannacry_payload_str2: www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
0x1a2cb300:$wannacry_payload_str2: www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
0x1b644cd8:$wannacry_payload_str2: www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
0x1d15945b:$wannacry_payload_str2: www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
0x1dd65c9f:$wannacry_payload_str2: www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com
0x450b048:$wannacry_payload_str3: msseccsvc.exe
0x5a7f3d4:$wannacry_payload_str3: msseccsvc.exe
0xda1c350:$wannacry_payload_str3: msseccsvc.exe
0x12481048:$wannacry_payload_str3: msseccsvc.exe
0x17027910:$wannacry_payload_str3: msseccsvc.exe
0x17f0dc18:$wannacry_payload_str3: msseccsvc.exe
```

- Beyond standalone tools, diving deeper into memory forensics offers a plethora of avenues.
 - Integrating YARA within memory forensics frameworks amplifies its potential.
 - With the Volatility framework and YARA operating in tandem, WannaCry-specific IOCs can be detected seamlessly.

- The [Volatility framework](#) is a powerful open-source memory forensics tool used to analyze memory images from various operating systems.
 - YARA can be integrated into the Volatility framework as a plugin called `yarascan` allowing for the application of YARA rules to memory analysis.
- The Volatility framework is covered in detail inside HTB Academy's [Introduction to Digital Forensics](#) module.
- For now, let's only discuss how YARA can be used as a plugin in the Volatility framework.

Single Pattern YARA Scanning Against a Memory Image

- In this case, we'll specify a YARA rule pattern directly in the command-line which is searched within the memory image by the `yarascan` plugin of Volatility.
 - The string should be enclosed in quotes (") after the `-U` option.
 - This is useful when we have a specific YARA rule or pattern that we want to apply without creating a separate YARA rules file.
- From previous analysis we know that WannaCry malware attempt to connect to the following hard-coded URI `www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com`
- Introducing this pattern within the command line using `-U "www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com"` prompts a search within the `compromised_system.raw` memory image.

```
areaeric@htb[/htb]$ vol.py -f /home/htb-
student/MemoryDumps/compromised_system.raw yarascan -U
"www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com"
```

Hunting Evil with YARA (Linux Edition)

```
areaeric@htb[/htb]$ vol.py -f /home/htb-student/MemoryDumps/compromised_system.raw yarascan -U "www.iuqe
Volatility Foundation Volatility Framework 2.6.1
/usr/local/lib/python2.7/dist-packages/volatility/plugins/community/YingLi/ssh_agent_key.py:12: Cryptogr
    from cryptography.hazmat.backends.openssl import backend
Rule: r1
Owner: Process svchost.exe Pid 1576
0x004313d7 77 77 77 2e 69 75 71 65 72 66 73 6f 64 70 39 69 www.iuquerfsodp9i
0x004313e7 66 6a 61 70 6f 73 64 66 6a 68 67 6f 73 75 72 69 fjaposdfjhgosuri
0x004313f7 6a 66 61 65 77 72 77 65 72 67 77 65 61 2e 63 6f jfaewrwerwgwea.co
0x00431407 6d 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 m.....
0x00431417 00 f0 5d 17 00 ff ff ff ff 00 00 00 00 00 00 00 00 00 ..].
0x00431427 00 00 00 00 00 00 00 00 20 00 00 00 04 00 00 .....
0x00431437 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x00431447 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x00431457 00 00 00 00 00 50 51 17 00 00 00 00 00 00 00 00 00 .....PQ..... .
0x00431467 00 13 00 00 00 b8 43 03 00 00 00 00 00 00 00 00 00 .....C..... .
0x00431477 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x00431487 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x00431497 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x004314a7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x004314b7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x004314c7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
Rule: r1
Owner: Process svchost.exe Pid 1576
0x0013dcfd 77 77 77 2e 69 75 71 65 72 66 73 6f 64 70 39 69 www.iuquerfsodp9i
0x0013dcfe 66 6a 61 70 6f 73 64 66 6a 68 67 6f 73 75 72 69 fjaposdfjhgosuri
0x0013dcff 6a 66 61 65 77 72 77 65 72 67 77 65 61 2e 63 6f jfaewrwerwgwea.co
0x0013dd08 6d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 m.....
---SNIP---
```

- This option allows us to directly specify a YARA rule string within the command-line itself.
 - Let's see how we can search for the content of a whole YARA rule file (i.e. `.yar` rule file) in memory image files.

Multiple YARA Rule Scanning Against a Memory Image

- When we have multiple YARA rules or a set of complex rules that we want to apply to a memory image, we can use the `-y` option followed by the rule file path in the Volatility framework, which allows us to specify the path to a YARA rules file.
 - The YARA rules file (`wannacry_artifacts_memory.yar` in our case) should contain one or more YARA rules in a separate file.
 - The YARA rules file we will use for demonstration purposes is the following.

```
areaeric@htb[/htb]$ cat /home/htb-
student/Rules/yara/wannacry_artifacts_memory.yar
```

```
areaeric@htb[/htb]$ cat /home/htb-student/Rules/yara/wannacry_artifacts_memory.yar
rule Ransomware_WannaCry {

meta:
    author = "Madhukar Raina"
    version = "1.1"
    description = "Simple rule to detect strings from WannaCry ransomware"
    reference = "https://www.virustotal.com/gui/file/ed01ebfb9eb5bbea545af4d01bf5f1071661840480439c

strings:
    $wannacry_payload_str1 = "tasksche.exe" fullword ascii
    $wannacry_payload_str2 = "www.iuquerfsodp9ifjaposdfjhgosurijfaewrwegwea.com" ascii
    $wannacry_payload_str3 = "mssecsvc.exe" fullword ascii
    $wannacry_payload_str4 = "diskpart.exe" fullword ascii
    $wannacry_payload_str5 = "lhdfrgui.exe" fullword ascii

condition:
    3 of them
```

- Let's run Volatility with the rule `wannacry_artifacts_memory.yar` (residing in the `/home/htb-student/Rules/yara` directory) to scan the memory image `compromised_system.raw` (residing in the `/home/htb-student/MemoryDumps` directory)

```
areaeric@htb[/htb]$ vol.py -f /home/htb-student/MemoryDumps/compromised_system.raw yarascan -y /home/htb-student/Rules/yara/wannacry_artifacts_memory.yar
```

Hunting Evil with YARA (Linux Edition)

```
areaeric@htb[/htb]$ vol.py -f /home/htb-student/MemoryDumps/compromised_system.raw yarascan -y /home/httpd/evil_yara.yar
Volatility Foundation Volatility Framework 2.6.1
/usr/local/lib/python2.7/dist-packages/volatility/plugins/community/YingLi/ssh_agent_key.py:12: Cryptogr
  from cryptography.hazmat.backends.openssl import backend
Rule: Ransomware_WannaCry
Owner: Process svchost.exe Pid 1576
0x0043136c 74 61 73 6b 73 63 68 65 2e 65 78 65 00 00 00 00 00 tasksche.exe....
0x0043137c 52 00 00 00 43 6c 6f 73 65 48 61 6e 64 6c 65 00 R...CloseHandle.
0x0043138c 57 72 69 74 65 46 69 6c 65 00 00 00 43 72 65 61 WriteFile...Crea
0x0043139c 74 65 46 69 6c 65 41 00 43 72 65 61 74 65 50 72 teFileA.CreatePr
0x004313ac 6f 63 65 73 73 41 00 00 6b 00 65 00 72 00 6e 00 ocessA..k.e.r.n.
0x004313bc 65 00 6c 00 33 00 32 00 2e 00 64 00 6c 00 6c 00 e.l.3.2...d.l.l.
0x004313cc 00 00 00 00 68 74 74 70 3a 2f 2f 77 77 77 2e 69 ....http://www.i
0x004313dc 75 71 65 72 66 73 6f 64 70 39 69 66 6a 61 70 6f uqerfsodp9ifjapo
0x004313ec 73 64 66 6a 68 67 6f 73 75 72 69 6a 66 61 65 77 sdfjhgosurijfaew
0x004313fc 72 77 65 72 67 77 65 61 2e 63 6f 6d 00 00 00 00 rwerwgwea.com....
0x0043140c 00 00 00 00 01 00 00 00 00 00 00 00 00 f0 5d 17 00 .....]...
0x0043141c ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x0043142c 00 00 00 00 20 00 00 00 04 00 00 00 01 00 00 00 ..... .
0x0043143c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x0043144c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0x0043145c 50 51 17 00 00 00 00 00 00 00 00 00 13 00 00 00 PQ.....
```

- We can see in the results that the `yarascan` plugin in Volatility is able to find the process `svchost.exe` with PID `1576` in the memory image of the compromised system.
- In summary, the `-U` option allows us to directly specify a YARA rule string within the command-line, while the `-y` option is used to specify the path to a file containing one or more YARA rules.
 - The choice between the two options depends on our specific requirements and whether we have a single rule or a set of rules to apply during the analysis.

One sentence summary

- We often only have memory dump/forensics of the compromised server, so volatile with yara becomes a good tool for this type of work.

Leveraging Sigma

Sigma and Sigma Rules

Overview

- `Sigma` is a generic signature format used for describing detection rules for log analysis and SIEM systems.
 - It allows SOC analysts to create and share rules that help identify specific patterns or behaviors indicative of security threats or malicious activities.
 - Sigma rules are typically written in YAML format and can be used with various security tools and platforms.
- SOC analysts use Sigma rules to define and detect security events by analyzing log data generated by various systems, such as firewalls, intrusion detection systems, and endpoint protection solutions.
 - These rules can be customized to match specific use cases and can include conditions, filters, and other parameters to determine when an event should trigger an alert.
- The main advantage of Sigma rules is their portability and compatibility with multiple SIEM and log analysis systems, enabling analysts to write rules once and use them across different platforms.
- Sigma can be considered as standardized format for analysts to create and share detection rules.
 - It helps in converting the IOCs into queries and can be easily integrated with security

tools, including SIEM and EDRs.

- Sigma rules can be used to detect suspicious activities in various log sources.
- This also helps in building efficient processes for Detection as Code by automating the creation and deployment of detection rules.



Usages of Sigma

- **Universal Log Analytics Tool**: We can write detection rules once and then convert them to various SIEM and log analytics tool formats, sparing us the repetitive task of rewriting logic across different platforms.
- **Community-driven Rule Sharing**: With Sigma, we have the ability to tap into a community that regularly contributes and shares their detection rules.
 - This ensures that we constantly update and refine our detection mechanisms.
- **Incident Response**: Sigma aids in incident response by enabling analysts to quickly search and analyze logs for specific patterns or indicators.
- **Proactive Threat Hunting**: We can use Sigma rules for proactive threat hunting sessions.
 - By leveraging specific patterns, we can comb through our datasets to pinpoint anomalies or signs of adversarial activity.
- **Seamless Integration with Automation Tools**: By converting Sigma rules into appropriate formats, we can seamlessly integrate them with our SOAR platforms and other automation tools, enabling automated responses based on specific detections.
- **Customization for Specific Environments**: The flexibility of Sigma rules means that we can tailor them according to the unique characteristics of our environment.
 - Custom rules can address the specific threats or scenarios we're concerned about.
- **Gap Identification**: By aligning our rule set with the broader community, we can perform gap analysis, identifying areas where our detection capabilities might need enhancement.

How Does Sigma Work?

- At its heart, Sigma is about expressing patterns found in log events in a structured manner.
 - So, instead of having a variety of rule descriptions scattered in various proprietary formats, with Sigma, we have a unified, open standard.
 - This unified format becomes the lingua franca for log-based threat detection.
- Sigma rules are written in YAML.
 - Each Sigma rule describes a particular pattern of log events which might correlate with malicious activity.
 - The rule encompasses a title, description, log source, and the pattern itself.
- But here comes the magic.
 - The true power of Sigma lies in its convertibility.
 - We might ask, "If it's a standard format, how do we use it with our specific logging tools and platforms?"
 - That's where the Sigma converter (`sigmac`) steps in.
 - This converter is the linchpin of our workflow, transforming our Sigma rules into queries or configurations compatible with a multitude of SIEMs, log management solutions, and other security analytics tools.
- With `sigmac`, we can take a rule written in the Sigma format and translate it for ElasticSearch, QRadar, Splunk, and many more, almost instantaneously.

Note: `pySigma` is increasingly becoming the go-to option for rule translation, as `sigmac` is now considered obsolete.

Sigma Rule Structure

- As mentioned already, Sigma rule files are written in YAML format. Below is the structure of a Sigma rule.

title	[required]
status	[optional]
description	[optional]
author	[optional]
reference	[optional]
...	
{arbitrary custom fields}	
logsource	[required]
category	[optional]
product	[optional]
service	[optional]
definition	[optional]
...	
{arbitrary custom fields}	
detection	[required]
{search-identifier}	[optional]
{string-list}	[optional]
{field: value}	[optional]
...	
timeframe	[optional]
condition	[required]
falsepositives	[optional]
level	[optional]
...	
{arbitrary custom fields}	

Source: <https://github.com/SigmaHQ/sigma/wiki/Specification>

- Let's understand the structure of a Sigma rule with an example.

Code: **yaml**

```
title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
description: Detects potential LethalHTA technique where "mshta.exe" is spawned by an "svchost.exe" process
references:
  - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
author: Markus Neis
date: 2018/06/07
tags:
  - attack.defense_evasion
  - attack.t1218.005
logsource:
  category: process_creation
  product: windows
detection:
  selection:
    ParentImage|endswith: '\svchost.exe'
    Image|endswith: '\mshta.exe'
  condition: selection
falsepositives:
  - Unknown
level: high
```

- The below screenshot shows the different components that form a Sigma Rule:

```

Title of the rule showing what the rule is supposed to detect
title: Potential LethalHTA Technique Execution

Globally Unique Identifier  
(randomly generated UUIDs)
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471

State of the rule (i.e. Stable, test, experimental, deprecated, unsupported)
status: test

More information on the objective of rule  
and the activity that can be detected
description: Detects potential LethalHTA technique where the "mshta.exe" is
spawned by an "svchost.exe" process
references:
  - https://codewhitesec.blogspot.com/2018/07/lethalhta.html

Author/Creator of the rule
author: Markus Neis

Date of creation
date: 2018/06/07

Tags: Context and information to categorize the rule.
tags:
  - attack.defense_evasion
  - attack.t1218.005

Describes the log data on which detection rule is meant to be applied to.
logsource:
  category: process_creation
  product: windows

Contains log source, platform, application and type required in the detection

Set of search-identifiers that represent properties of searches on log data
detection:
  selection:
    ParentImage|endswith: '\svchost.exe'
    Image|endswith: '\mshta.exe'
  condition: selection

known false positives that may occur
falsepositives:
  - Unknown

Describes the criticality of a triggered rule.
level: high
  (i.e. Informational, Low, medium, high and critical)

```

Sigma Rule Breakdown (based on Sigma's specification):

- title**: A brief title for the rule that should contain what the rule is supposed to detect (max. 256 characters)

Code: yaml

```

title: Potential LethalHTA Technique Execution
...

```

- id**: Sigma rules should be identified by a globally unique identifier in the id attribute.
- For this purpose randomly generated UUIDs (version 4) are recommended but not

mandatory.

Code: yaml

```
title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
...
...
```

- `status` (optional): Declares the status of the rule.
 - `stable`: The rule didn't produce any obvious false positives in multiple environments over a long period of time
 - `test`: The rule doesn't show any obvious false positives on a limited set of test systems
 - `experimental`: A new rule that hasn't been tested outside of lab environments and could lead to many false positives
 - `deprecated`: The rule is to replace or cover another one. The link between rules is made via the related field.
 - `unsupported`: The rule can not be used in its current state (special correlation log, home-made fields, etc.)

Code: yaml

```
title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
...
...
```

- `description` (optional): A short description of the rule and the malicious activity that can be detected (max. 65,535 characters)

Code: yaml

```
title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
description: Detects potential LethalHTA technique where "mshta.exe" is spawned by an "svchost.exe" proc
...
...
```

- `references` (optional): Citations to the original source from which the rule was inspired.

- These might include blog posts, academic articles, presentations, or even tweets.

Code: `yaml`

```
title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
description: Detects potential LethalHTA technique where "mshta.exe" is spawned by an "svchost.exe" process
references:
  - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
...
```

- `author` (optional): Creator of the rule (can be a name, nickname, twitter handle, etc).

Code: `yaml`

```
title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
description: Detects potential LethalHTA technique where "mshta.exe" is spawned by an "svchost.exe" process
references:
  - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
author: Markus Neis
...
```

- `date` (optional): Rule creation date. Use the format `YYYY/MM/DD`.

Code: `yaml`

```
title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
description: Detects potential LethalHTA technique where "mshta.exe" is spawned by an "svchost.exe" process
references:
  - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
author: Markus Neis
date: 2018/06/07
...
```

- `logsource`: This section describes the log data on which the detection is meant to be applied to. It describes the log source, the platform, the application and the type that is required in the detection.
 - More information can be found in the following link:
<https://github.com/SigmaHQ/sigma/tree/master/documentation/logsource-guides>
- It consists of three attributes that are evaluated automatically by the converters and an arbitrary number of optional elements.
 - We recommend using a "definition" value when further explanation is necessary.

- `category`: The `category` value is used to select all log files written by a certain group of products, like firewalls or web server logs.

- The automatic converter will use the keyword as a selector for multiple indices. Examples: `firewall`, `web`, `antivirus`, etc.

- `product`: The `product` value is used to select all log outputs of a certain product, e.g. all Windows event log types including `Security`, `System`, `Application` and newer types like `AppLocker` and `Windows Defender`. Examples: `windows`, `apache`, `check point fw1`, etc.
- `service`: The `service` value is used to select only a subset of a product's logs, like the `sshd` on Linux or the `Security` event log on Windows systems. Examples: `sshd`, `applocker`, etc.

Code: [yaml](#)

```

title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
description: Detects potential LethalHTA technique where "mshta.exe" is spawned by an "svchost.exe" process
references:
  - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
author: Markus Neis
date: 2018/06/07
logsource:
  category: process_creation
  product: windows
...

```

- **detection**: A set of search-identifiers that represent properties of searches on log data.
 - Detection is made up of two components:

- Search Identifiers
- Condition

```

1
2  ### Search Identifier, Condition Example
3
4  detection:
5    selection1:
6      Image|endswith:
7        - 'cmd.exe'
8        - 'powershell.exe'
9    selection2:
10   ParentImage|endswith:
11     - 'winword.exe'
12     - 'excel.exe'
13     - 'powerpnt.exe'
14   condition: selection1 AND selection2

```

Code: yaml

```

title: Potential LethalHTA Technique Execution
id: ed5d72a6-f8f4-479d-ba79-02f6a80d7471
status: test
description: Detects potential LethalHTA technique where "mshta.exe" is spawned by an "svchost.exe" process
references:
  - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
author: Markus Neis
date: 2018/06/07
logsource:
  category: process_creation
  product: windows
detection:
  selection:
    ParentImage|endswith: '\svchost.exe'
    Image|endswith: '\mshta.exe'
  condition: selection
...

```

- The values contained in Sigma rules can be modified by value modifiers.
 - Value modifiers are appended after the field name with a pipe character (|) as separator and can also be chained, e.g. `fieldname|mod1|mod2: value`.
 - The value modifiers are applied in the given order to the value.
- The behavior of search identifiers is changed by value modifiers as shown in the table below :

Value	Modifier	Explanation	Example
contains		Adds wildcard (*) characters around the value(s)	CommandLine contains
all		Links all elements of a list with a logical "AND" (instead of the default "OR")	CommandLine contains all
startswith		Adds a wildcard (*) character at the end of the field value	ParentImage startswith
endswith		Adds a wildcard (*) character at the beginning of the field value	Image endswith
re:		This value is handled as regular expression by backends	CommandLine re: '\[String \]\s*\\$VerbosePreference

Source: blusapphire.io

Search identifiers include multiple values in two different data structures:

1. `Lists`, which can contain:

- `strings` that are applied to the full log message and are linked with a logical `OR`.
- `maps` (see below). All map items of a list are linked with a logical `AND`.

```

2  ### Search Identifier Example, Type: List
3  detection:
4    selection:
5      Image|endswith:
6        - 'cmd.exe'
7        - 'powershell.exe'
8      ParentImage|endswith:
9        - 'winword.exe'
10       - 'excel.exe'
11       - 'powerpnt.exe'
12   condition: selection
13
14  ### Search Identifier Example, Type: Maps
15  detection:
16    selection:
17      Image|endswith: '\wmic.exe'
18      CommandLine|contains: '/node:'
19   condition: selection
20
21

```

List
Elements in a list begin with "-" dash bullet and are linked with logical 'OR'

Condition Matches:
(Image == 'cmd.exe' OR Image == 'powershell.exe') AND
(ParentImage == 'winword.exe' OR ParentImage == 'excel.exe' OR ParentImage == 'powerpnt.exe')

Maps (Key-Value Pair)
Key is the field name from the log data or event
Value can be String/Integer value searching for
Elements are linked with Logical 'AND'

Condition Matches:
(Image == 'wmic.exe' AND CommandLine == '/node:')

2. **Source:** blusapphire.io

- Example list of strings that matches on `evilservice` or `svchost.exe -n evil`.

Code: yaml

```
detection:
  keywords:
    - evilservice
    - svchost.exe -n evil
```

- Example list of maps that matches on image file `example.exe` or on a executable whose description contains the string `Test executable`.

Code: yaml

```
detection:
  selection:
    - Image|endswith: '\example.exe'
    - Description|contains: 'Test executable'
```

2. Maps : Maps (or dictionaries) consist of key/value pairs, in which the key is a field in the log data and the value a string or integer value.

- All elements of a map are joined with a logical `AND`.
- Example that matches on event log `Security` and (`Event ID 517` or `Event ID 1102`)

Code: yaml

```
detection:
  selection:
    EventLog: Security
    EventID:
      - 517
      - 1102
  condition: selection
```

- Example that matches on event log `Security` and `Event ID 4679` and `TicketOptions 0x40810000` and `TicketEncryption 0x17`.

Code: yaml

```
detection:  
    selection:  
        EventLog: Security  
        EventID: 4769  
        TicketOptions: '0x40810000'  
        TicketEncryption: '0x17'  
    condition: selection
```

- **Condition:** Condition defines how fields are related to each other. If there's anything to filter, it can be defined in the condition.
 - It uses various operators to define relationships for multiple fields which are explained below.

Operator	Example
Logical AND/OR	<code>keywords1 or keywords2</code>
1/all of them	<code>all of them</code>
1/all of search-identifier-pattern	<code>all of selection*</code>
1/all of search-id-pattern	<code>all of filter_*</code>
Negation with 'not'	<code>keywords and not filters</code>
Brackets - Order of operation '()'	<code>selection1 and (keywords1 or keywords2)</code>

Source: blusapphire.io

- Example of a condition:

Code: yaml

```
condition: selection1 or selection2 or selection3
```

Sigma Rule Development Best Practices

- Sigma's specification repository contains everything you may need around Sigma rules.
- Sigma rule development best practices and common pitfalls can be found on [Sigma's Rule Creation Guide](#).

One sentence summary

- Sigma is a tool for standardizing and converting query to various SIEM platform.

Developing Sigma Rules

Manually Developing a Sigma Rule

Example 1: LSASS Credential Dumping

- Let's dive into the world of Sigma rules using a sample named `shell.exe` (a renamed version of [mimikatz](#)) residing in the `C:\Samples\YARASigma` directory of this section's target as an illustration.
 - We want to understand the process behind crafting a Sigma rule, so let's get our hands dirty.
- After executing `shell.exe` as follows, we collected the most critical events and saved them as `lab_events.evtx` inside the `C:\Events\YARASigma` directory of this section's target.
- The process created by `shell.exe` ([mimikatz](#)) will try to access the process memory of `lsass.exe`.
 - The system monitoring tool [Sysmon](#) was running in the background and captured this activity in the event logs (Event ID [10](#)).

```
C:\Samples\YARASigma>shell.exe

mimikatz # privilege::debug

mimikatz # sekurlsa::logonpasswords
```

```
C:\Samples\YARASigma>shell.exe

.#####. mimikatz 2.2.0 (x64) #19041 Sep 19 2022 17:44:08
.## ^ ##. "A La Vie, A L'Amour" - (oe.eo)
## / \ ## /** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##      > https://blog.gentilkiwi.com/mimikatz
## v ##'      Vincent LE TOUX          ( vincent.letoux@gmail.com )
'#####'      > https://pingcastle.com / https://mysmartlogon.com **/


mimikatz # privilege::debug
Privilege '20' OK

mimikatz # sekurlsa::logonpasswords
---SNIP---
Authentication Id : 0 ; 100080 (00000000:000186f0)
Session           : Interactive from 1
User Name         : htb-student
Domain            : DESKTOP-VJF8GH8
Logon Server     : DESKTOP-VJF8GH8
Logon Time       : 8/25/2023 2:17:20 PM
SID               : S-1-5-21-1412399592-1502967738-1150298762-1001
msv :
[00000003] Primary
* Username : htb-student
* Domain   : .
* NTLM     : 3c0e5d303ec84884ad5c3b7876a06ea6
* SHA1     : b2978f9abc2f356e45cb66ec39510b1ccca08a0e
tspkg :
```

- First off, Sysmon Event ID 10 is triggered when a process accesses another process, and it logs the permission flags in the GrantedAccess field.
 - This event log contains two important fields, TargetImage and GrantedAccess.
 - In a typical LSASS memory dumping scenario, the malicious process needs specific permissions to access the memory space of the LSASS process.

- These permissions are often read/write access, among other things.

The screenshot shows the 'Event 10, Sysmon' window with the 'General' tab selected. The main pane displays the following log entry:

```

Process accessed:
RuleName:
UtcTime: 2023-07-09 14:44:14.260
SourceProcessGUID: {e7bf76b7-c7ba-64aa-0000-0010e8e9a602}
SourceProcessId: 1884
SourceThreadId: 7872
SourceImage: C:\htb\samples\shell.exe
TargetProcessGUID: {e7bf76b7-d7ec-6496-0000-001027d60000}
TargetProcessId: 668
TargetImage: C:\Windows\system32\lsass.exe
GrantedAccess: 0x1010
CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9d4c4|C:\Windows\System32\KERNELBASE.dll+2c13e|C:\htb\samples\shell.exe+c291e|C:\htb\samples\shell.exe+c2cf5|C:\htb\samples\shell.exe+c285d|C:\htb\samples\shell.exe+85a44|C:\htb\samples\shell.exe+8587c|C:\htb\samples\shell.exe+85647|C:\htb\samples\shell.exe+c97a5|C:\Windows\System32\KERNEL32.DLL+17034|C:\Windows\SYSTEM32\ntdll.dll+526a1

```

Below the log entry, the event details are summarized:

Log Name:	Microsoft-Windows-Sysmon/Operational		
Source:	Sysmon	Logged:	09-07-2023 20:14:14
Event ID:	10	Task Category:	Process accessed (rule: ProcessAccess)
Level:	Information	Keywords:	
User:	SYSTEM	Computer:	RDSEMVM01
OpCode:	Info		
More Information: Event Log Online Help			

- Now, why is `0x1010` crucial here?
 - This hexadecimal flag essentially combines `PROCESS_VM_READ` (`0x0010`) and `PROCESS_QUERY_INFORMATION` (`0x0400`) permissions.
 - To translate that: the process is asking for read access to the virtual memory of LSASS and the ability to query certain information from the process.
 - While `0x0410` is the most common GrantedAccess flag used for reading LSASS memory, `0x1010` implies both reading and querying information from the process and is also frequently observed during credential dumping attacks.
- So how can we weaponize this information for detection?
 - Well, in our security monitoring stack, we would configure Sysmon to flag or alert on any `Event ID 10` where the `TargetImage` is `lsass.exe` and `GrantedAccess` is set to `0x1010`.

- A Sigma rule that checks for the abovementioned conditions can be found below.

Code: yaml

```
title: LSASS Access with rare GrantedAccess flag
status: experimental
description: This rule will detect when a process tries to access LSASS memory with suspicious access fl
date: 2023/07/08
tags:
  - attack.credential_access
  - attack.t1003.001
logsource:
  category: process_access
  product: windows
detection:
  selection:
    TargetImage|endswith: '\lsass.exe'
    GrantedAccess|endswith: '0x1010'
  condition: selection
```

Sigma Rule Breakdown

- `title`: This title offers a concise overview of the rule's objective, specifically aimed at detecting interactions with LSASS memory involving a particular access flag.

Code: yaml

```
title: LSASS Access with rare GrantedAccess flag
```

- `status`: This field signals that the rule is in the testing phase, suggesting that additional fine-tuning or validation may be necessary.

Code: yaml

```
status: experimental
```

- `description`: Rule description.

`description`: This rule will detect when a process tries to access LSASS memory with suspicious access flag 0x1010

- `date`: This field marks the date when the rule was either updated or originally created.

`date`: 2023/07/08

- `tags`: The rule is tagged with `attack.credential_access` and `attack.t1003.001`.

- These tags help categorize the rule based on known attack techniques or tactics related to credential access.

```
tags:
  - attack.credential_access
  - attack.t1003.001`
```

- `logsource` : The logsource specifies the log source that the rule is intended to analyze.
 - It contains `category` as `process_access` which indicates that the rule focuses on log events related to process access (Sysmon Event ID 10 , if we use Sigma's default config files).
 - Also, `product: windows` specifies that the rule is specifically designed for Windows operating systems.

Code: yaml

```
logsource:
  category: process_access
  product: windows
```

- `detection` : The detection section defines the conditions that must be met for the rule to trigger an alert.
 - The selection part specifies the criteria for selecting relevant log events where the `TargetImage` field ends with `\lsass.exe` and `GrantedAccess` field ends with the hexadecimal value `0x1010` .
 - The `GrantedAccess` field represents the access rights or permissions associated with the process.
 - In this case, it targets events with a specific access flag of `0x1010` .
 - Finally, the condition part specifies that the selection criteria must be met for the rule to trigger an alert.
 - In this case, both the `TargetImage` and `GrantedAccess` criteria must be met.

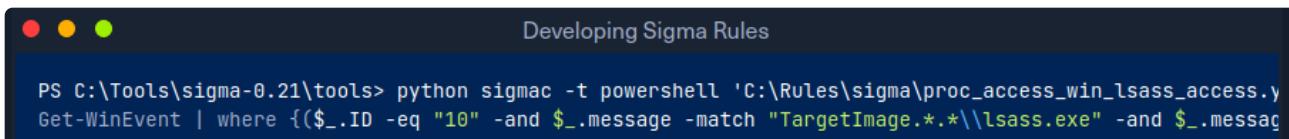
Code: yaml

```
detection:
  selection:
    TargetImage|endswith: '\lsass.exe'
    GrantedAccess|endswith: '0x1010'
  condition: selection
```

- Our first Sigma rule above can be found inside the `C:\Rules\sigma` directory of this section's target as `proc_access_win_lsass_access.yml`.
 - Let's explore the `sigmac` tool that can help us transform this rule into queries or configurations compatible with a multitude of SIEMs, log management solutions, and other security analytics tools.
- The `sigmac` tool can be found inside the `C:\Tools\sigma-0.21\tools` directory of this section's target.
- Suppose that we wanted to convert our Sigma rule into a PowerShell (`Get-WinEvent`) query.
 - This could have been accomplished with the help of `sigmac` as follows.

```
PS C:\Tools\sigma-0.21\tools> python sigmac -t powershell
'C:\Rules\sigma\proc_access_win_lsass_access.yml'

Get-WinEvent | where {($_.ID -eq "10" -and $_.message -match
"TargetImage.*.*\\lsass.exe" -and $_.message -match
"GrantedAccess.*.*0x1010") } | select
TimeCreated,Id,RecordId,ProcessId,MachineName,Message
```



```
PS C:\Tools\sigma-0.21\tools> python sigmac -t powershell 'C:\Rules\sigma\proc_access_win_lsass_access.yml'
Get-WinEvent | where {($_.ID -eq "10" -and $_.message -match "TargetImage.*.*\\lsass.exe" -and $_.message -match "GrantedAccess.*.*0x1010") } | select
TimeCreated,Id,RecordId,ProcessId,MachineName,Message
```

- Let's adjust the Get-WinEvent query above by specifying the .evtx file that is related to LSASS access by another process (`lab_events.evtx` inside the `C:\Events\YARASigma` directory of this section's target) and see if it will identify the Sysmon event (ID 10) that we analyzed at the beginning of this section.

Note: Please open a PowerShell terminal as administrator to run the query.

```
PS C:\Tools\sigma-0.21\tools> Get-WinEvent -Path
C:\Events\YARASigma\lab_events.evtx | where {($_.ID -eq "10" -and
$_.message -match "TargetImage.*.*\\lsass.exe" -and $_.message -match
"GrantedAccess.*.*0x1010") } | select
TimeCreated,Id,RecordId,ProcessId,MachineName,Message
```

```
Developing Sigma Rules

PS C:\Tools\sigma-0.21\tools> Get-WinEvent -Path C:\Events\YARASigma\lab_events.evtx | where {($_.ID -eq 10) -and ($_.Message -like "Process accessed")}

TimeCreated : 7/9/2023 7:44:14 AM
Id          : 10
RecordId    : 7810
ProcessId   : 3324
MachineName : RDSEMVM01
Message     : Process accessed:
  RuleName:
    Utctime: 2023-07-09 14:44:14.260
    SourceProcessGUID: {e7bf76b7-c7ba-64aa-0000-0010e8e9a602}
    SourceProcessId: 1884
    SourceThreadId: 7872
    SourceImage: C:\htb\samples\shell.exe
    TargetProcessGUID: {e7bf76b7-d7ec-6496-0000-001027d60000}
    TargetProcessId: 668
    TargetImage: C:\Windows\system32\lsass.exe
    GrantedAccess: 0x1010
    CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9d4c4|C:\Windows\System32\KERNELBASE.dll+2c13e|C:\ell.exe+c291e|C:\htb\samples\shell.exe+c2cf5|C:\htb\samples\shell.exe+c285d|C:\htb\samples4|C:\htb\samples\shell.exe+8587c|C:\htb\samples\shell.exe+85647|C:\htb\samples\shell.exe+c\SYSTEM32\KERNEL32.DLL+17034|C:\Windows\SYSTEM32\ntdll.dll+526a1
    SourceUser: %12
    TargetUser: %13
```

- The related Sysmon event with ID 10 is successfully identified!
- But let's not stop there - remember, false positives are the enemy of effective security monitoring.
 - We should also cross-reference the `SourceImage` (the process initiating the access) against a list of known, safe processes that commonly interact with LSASS.
 - If we see an unfamiliar or unusual process trying to read LSASS with a `GrantedAccess` that ends with `10`, `30`, `50`, `70`, `90`, `B0`, `D0`, `F0`, `18`, `38`, `58`, `78`, `98`, `B8`, `D8`, `F8`, `1A`, `3A`, `5A`, `7A`, `9A`, `BA`, `DA`, `FA`, `0x14C2`, and `FF` (these suffixes come from studying the `GrantedAccess` values that various LSASS credential dumping techniques require), that's a red flag, and our incident response protocol should kick in.
 - Especially, if the `SourceImage` resides in suspicious paths containing, `\Temp\`, `\Users\Public\`, `\PerfLogs\`, `\AppData\`, `\htb\` etc. that's another red flag, and our incident response protocol should kick in.
- A more robust version of the Sigma rule we created taking the above points into consideration can be found inside the `C:\Rules\sigma` directory of this section's target as `proc_access_win_lsass_access_robust.yml`

```
title: LSASS Access From Program in Potentially Suspicious Folder
id: fa34b441-961a-42fa-a100-ecc28c886725
```

```
status: experimental
description: Detects process access to LSASS memory with suspicious
access flags and from a potentially suspicious folder
references:
  - https://docs.microsoft.com/en-us/windows/win32/procthread/process-
    security-and-access-rights
  - https://onedrive.live.com/view.aspx?
resid=D026B4699190F1E6!2843&ithint=file%2cpptx&app=PowerPoint&authkey=!A
MvCRTKB_V1J5ow
  -
  https://web.archive.org/web/20230208123920/https://cyberwardog.blogspot.
  com/2017/03/chronicles-of-threat-hunter-hunting-for_22.html
  - https://www.slideshare.net/heirhabarov/hunting-for-credentials-
    dumping-in-windows-environment
  - http://security-research.dyndns.org/pub/slides/FIRST2017/FIRST-
    2017_Tom-Ueltschi_Sysmon_FINAL_notes.pdf
author: Florian Roth (Nextron Systems)
date: 2021/11/27
modified: 2023/05/05
tags:
  - attack.credential_access
  - attack.t1003.001
  - attack.s0002
logsource:
  category: process_access
  product: windows
detection:
  selection:
    TargetImage|endswith: '\lsass.exe'
    GrantedAccess|endswith:
      - '10'
      - '30'
      - '50'
      - '70'
      - '90'
      - 'B0'
      - 'D0'
      - 'F0'
      - '18'
      - '38'
      - '58'
      - '78'
      - '98'
      - 'B8'
      - 'D8'
      - 'F8'
```

```
- '1A'
- '3A'
- '5A'
- '7A'
- '9A'
- 'BA'
- 'DA'
- 'FA'
- '0x14C2'  #
https://github.com/b4rtik/ATPMiniDump/blob/76304f93b390af3bb66e4f451ca16562a479bdc9/ATPMiniDump/ATPMiniDump.c
- 'FF'

SourceImage|contains:
- '\Temp\
- '\Users\Public\
- '\PerfLogs\
- '\AppData\
- '\htb\'

filter_optional_generic_appdata:
SourceImage|startswith: 'C:\Users\
SourceImage|contains: '\AppData\Local\
SourceImage|endswith:
- '\Microsoft VS Code\Code.exe'
- '\software_reporter_tool.exe'
- '\DropboxUpdate.exe'
- '\MBAMInstallerService.exe'
- '\WebexMTA.exe'
- '\WebEx\WebexHost.exe'
- '\JetBrains\Toolbox\bin\jetbrains-toolbox.exe'

GrantedAccess: '0x410'

filter_optional_dropbox_1:
SourceImage|startswith: 'C:\Windows\Temp\
SourceImage|endswith: '.tmp\DropboxUpdate.exe'
GrantedAccess:
- '0x410'
- '0x1410'

filter_optional_dropbox_2:
SourceImage|startswith: 'C:\Users\
SourceImage|contains: '\AppData\Local\Temp\
SourceImage|endswith: '.tmp\DropboxUpdate.exe'
GrantedAccess: '0x1410'

filter_optional_dropbox_3:
SourceImage|startswith:
- 'C:\Program Files (x86)\Dropbox\
- 'C:\Program Files\Dropbox\
SourceImage|endswith: '\DropboxUpdate.exe'
```

```
GrantedAccess: '0x1410'
filter_optional_nextron:
    SourceImage|startswith:
        - 'C:\Windows\Temp\asgard2-agent\' 
        - 'C:\Windows\Temp\asgard2-agent-sc\' 
    SourceImage|endswith:
        - '\thor64.exe' 
        - '\thor.exe' 
        - '\aurora-agent-64.exe' 
        - '\aurora-agent.exe' 
    GrantedAccess:
        - '0xfffffff' 
        - '0x1010' 
        - '0x101010' 
filter_optional_ms_products:
    SourceImage|startswith: 'C:\Users\' 
    SourceImage|contains|all:
        - '\AppData\Local\Temp\' 
        - '\vs_bootstrapper_\' 
    GrantedAccess: '0x1410' 
filter_optional_chrome_update:
    SourceImage|startswith: 'C:\Program Files (x86)\Google\Temp\' 
    SourceImage|endswith: '.tmp\GoogleUpdate.exe' 
    GrantedAccess:
        - '0x410' 
        - '0x1410' 
filter_optional_keybase:
    SourceImage|startswith: 'C:\Users\' 
    SourceImage|endswith: '\AppData\Local\Keybase\keybase.exe' 
    GrantedAccess: '0xfffffff' 
filter_optional_avira:
    SourceImage|contains: '\AppData\Local\Temp\is-' 
    SourceImage|endswith: '.tmp\avira_system_speedup.tmp' 
    GrantedAccess: '0x1410' 
filter_optional_viberpc_updater:
    SourceImage|startswith: 'C:\Users\' 
    SourceImage|contains: '\AppData\Roaming\ViberPC\' 
    SourceImage|endswith: '\update.exe' 
    TargetImage|endswith: '\winlogon.exe' 
    GrantedAccess: '0xfffffff' 
filter_optional_adobe_arm_helper:
    SourceImage|startswith: # Example path: 'C:\Program Files (x86)\Common Files\Adobe\ARM\1.0\Temp\2092867405\AdobeARMHelper.exe' 
        - 'C:\Program Files\Common Files\Adobe\ARM\' 
        - 'C:\Program Files (x86)\Common Files\Adobe\ARM\' 
    SourceImage|endswith: '\AdobeARMHelper.exe'
```

```
GrantedAccess: '0x1410'
condition: selection and not 1 of filter_optional_*
fields:
- User
- SourceImage
- GrantedAccess
falsepositives:
- Updaters and installers are typical false positives. Apply custom
filters depending on your environment
level: medium
```

- Notice how the condition filters out false positives (selection and not 1 of filter_optional_*).

Example 2: Multiple Failed Logins From Single Source (Based on Event 4776)

- According to Microsoft, [Event 4776](#) generates every time that a credential validation occurs using NTLM authentication.
- This event occurs only on the computer that is authoritative for the provided credentials.
 - For domain accounts, the domain controller is authoritative.
 - For local accounts, the local computer is authoritative.
- It shows successful and unsuccessful credential validation attempts.
- It shows only the computer name (`Source Workstation`) from which the authentication attempt was performed (authentication source).
 - For example, if you authenticate from CLIENT-1 to SERVER-1 using a domain account you'll see CLIENT-1 in the `Source Workstation` field.
 - Information about the destination computer (SERVER-1) isn't presented in this event.
- If a credential validation attempt fails, you'll see a Failure event with Error Code parameter value not equal to `0x0`.
- `lab_events_2.evtx` inside the `C:\Events\YARASigma` directory of this section's target contains events related to multiple failed login attempts against `NOUSER` (thanks to [mdecrevoisier](#)).

lab_events_2 Number of events: 20

Level	Date and Time	Source	Event ID	Task Ca...
Information	5/20/2021 5:49:46 AM	Micros...	4717	Authen...
Information	5/20/2021 5:49:46 AM	Micros...	4718	Authen...
Information	5/20/2021 5:49:52 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:54 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:53 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:54 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:54 AM	Micros...	4776	Creden...

Event 4776, Microsoft Windows security auditing.

[General](#) [Details](#)

The computer attempted to validate the credentials for an account.

Authentication Package: MICROSOFT_AUTHENTICATION_PACKAGE_V1_0
Logon Account: NOUSER
Source Workstation: FS01
Error Code: 0xC0000064

Log Name: Security
Source: Microsoft Windows security
Event ID: 4776
Level: Information
User: N/A
OpCode: Info
More Information: [Event Log Online Help](#)



lab_events_2 Number of events: 20

Level	Date and Time	Source	Event ID	Task Ca...
Information	5/20/2021 5:49:46 AM	Micros...	4717	Authen...
Information	5/20/2021 5:49:46 AM	Micros...	4718	Authen...
Information	5/20/2021 5:49:52 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:54 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:53 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:54 AM	Micros...	4776	Creden...
Information	5/20/2021 5:49:54 AM	Micros...	4776	Creden...

Event 4776, Microsoft Windows security auditing.



General Details

Friendly View XML View

+ System

- EventData

PackageName MICROSOFT_AUTHENTICATION_PACKAGE_V

TargetUserName NOUSER

Workstation FS01

Status 0xc0000064

- A valid Sigma rule to detect multiple failed login attempts originating from the same source can be found inside the `C:\Rules\sigma` directory of this section's target, saved as `win_security_susp_failed_logons_single_source2.yml`

Code: yaml

```
title: Failed NTLM Logins with Different Accounts from Single Source System
id: 6309ffc4-8fa2-47cf-96b8-a2f72e58e538
related:
  - id: e98374a6-e2d9-4076-9b5c-11bdb2569995
    type: derived
status: unsupported
description: Detects suspicious failed logins with different user accounts from a single source system
author: Florian Roth (Nextron Systems)
date: 2017/01/10
modified: 2023/02/24
tags:
  - attack.persistence
  - attack.privilege_escalation
  - attack.t1078
logsource:
  product: windows
  service: security
detection:
  selection2:
    EventID: 4776
    TargetUserName: '*'
    Workstation: '*'
  condition: selection2 | count(TargetUserName) by Workstation > 3
falsepositives:
  - Terminal servers
  - Jump servers
  - Other multiuser systems like Citrix server farms
  - Workstations with frequently changing users
level: medium
```

Sigma Rule Breakdown:

- `logsource` : This section specifies that the rule is intended for Windows systems (`product: windows`) and focuses only on `Security` event logs (`service: security`).

Code: yaml

```
logsource:
  product: windows
  service: security
```

- `detection: selection2` is essentially the filter. It's looking for logs with `EventID 4776` (`EventID: 4776`) regardless of the `TargetUserName` or `Workstation` values (`TargetUserName: '*'`, `Workstation: '*'`).
 - `condition` counts instances of `TargetUserName` grouped by `Workstation` and checks if a workstation has more than `three` failed login attempts.

- As you can imagine, the best Sigma rule development resource is the official documentation, which can be found at the following links.
 - <https://github.com/SigmaHQ/sigma/wiki/Rule-Creation-Guide>
 - <https://github.com/SigmaHQ/sigma-specification>
- The following series of articles is the next best resource on Sigma rule development.
 - <https://tech-en.netlify.app/articles/en510480/>
 - <https://tech-en.netlify.app/articles/en513032/>
 - <https://tech-en.netlify.app/articles/en515532/>

One sentence summary

- Illustrate how we can convert sigma rules to powershell Get-WinEvent queries.

Hunting Evil with Sigma (Chainsaw Edition)

Summary

- In cybersecurity, time is of the essence.
 - Rapid analysis allows us to not just identify but also respond to threats before they escalate.
- When we're up against the clock, racing to find a needle in a haystack of Windows Event Logs without access to a SIEM, Sigma rules combined with tools like [Chainsaw](#) and [Zircolite](#) are our best allies.
- Both tools allow us to use Sigma rules to scan not just one, but multiple EVTX files concurrently, offering a broader and more comprehensive scan in a very efficient manner.

Scanning Windows Event Logs With Chainsaw

- Chainsaw is a freely available tool designed to swiftly pinpoint security threats within Windows Event Logs.
 - This tool enables efficient keyword-based event log searches and is equipped with integrated support for Sigma detection rules as well as custom Chainsaw rules.
 - Therefore, it serves as a valuable asset for validating our Sigma rules by applying them to actual event logs.
 - Let's download the Chainsaw from the official Github repository and run it with some sigma rules:

- Chainsaw can be found inside the `C:\Tools\chainsaw` directory of this section's target.
- Let's first run Chainsaw with `-h` flag to see the help menu.

```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe -h
```

```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe -h
Rapidly work with Forensic Artefacts

Usage: chainsaw_x86_64-pc-windows-msvc.exe [OPTIONS] <COMMAND>

Commands:
  dump      Dump an artefact into a different format
  hunt      Hunt through artefacts using detection rules for threat detection
  lint      Lint provided rules to ensure that they load correctly
  search    Search through forensic artefacts for keywords
  analyse   Perform various analyses on artifacts
  help      Print this message or the help of the given subcommand(s)

Options:
  --no-banner          Hide Chainsaw's banner
  --num-threads <NUM_THREADS> Limit the thread number (default: num of CPUs)
  -h, --help            Print help
  -V, --version         Print version

Examples:

  Hunt with Sigma and Chainsaw Rules:
  ./chainsaw hunt evtx_attack_samples/ -s sigma/ --mapping mappings/sigma-event-logs-all.yml -r ru

  Hunt with Sigma rules and output in JSON:
  ./chainsaw hunt evtx_attack_samples/ -s sigma/ --mapping mappings/sigma-event-logs-all.yml --jsco

  Search for the case-insensitive word 'mimikatz':
  ./chainsaw search mimikatz -i evtx_attack_samples/

  Search for Powershell Script Block Events (EventID 4014):
  ./chainsaw search -t 'Event.System.EventID: =4104' evtx_attack_samples/
```

Example 1: Hunting for Multiple Failed Logins From Single Source With Sigma

- Let's put Chainsaw to work by applying our most recent Sigma rule, `win_security_susp_failed_logons_single_source2.yml` (available at `C:\Rules\sigma`), to `lab_events_2.evtx` (available at `C:\Events\YARASigma\lab_events_2.evtx`) that contains multiple failed login attempts from the same source.

```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe hunt
C:\Events\YARASigma\lab_events_2.evtx -s
C:\Rules\sigma\win_security_susp_failed_logons_single_source2.yml --mapping .\mappings\sigma-event-logs-all.yml
```

- Our Sigma rule was able to identify the multiple failed login attempts against `NOUSER`.
- Using the `-s` parameter, we can specify a directory containing Sigma detection rules (or one Sigma detection rule) and Chainsaw will automatically load, convert and run these rules against the provided event logs.
 - The mapping file (specified through the `--mapping` parameter) tells Chainsaw which fields in the event logs to use for rule matching.

Example 2: Hunting for Abnormal PowerShell Command Line Size With Sigma (Based on Event ID 4688)

- Firstly, let's set the stage by recognizing that PowerShell, being a highly flexible scripting language, is an attractive target for attackers.
 - Its deep integration with Windows APIs and .NET Framework makes it an ideal candidate for a variety of post-exploitation activities.
- To conceal their actions, attackers utilize complex encoding layers or misuse cmdlets for purposes they weren't designed for.
 - This leads to abnormally long PowerShell commands that often incorporate Base64 encoding, string merging, and several variables containing fragmented parts of the command.
- A Sigma rule that can detect abnormally long PowerShell command lines can be found inside the `C:\Rules\sigma` directory of this section's target, saved as `proc_creation_win_powershell_abnormal_commandline_size.yml`.

```

title: Unusually Long PowerShell CommandLine
id: d0d28567-4b9a-45e2-8bbc-fb1b66a1f7f6
status: test
description: Detects unusually long PowerShell command lines with a length of 1000 characters or more
references:
  - https://speakerdeck.com/heirhabarov/hunting-for-powershell-abuse
author: oscd.community, Natalia Shornikova / HTB Academy, Dimitrios Bougioukas
date: 2020/10/06
modified: 2023/04/14
tags:
  - attack.execution
  - attack.t1059.001
  - detection.threat_hunting
logsource:
  category: process_creation
  product: windows

```

```
detection:
  selection:
    EventID: 4688
    NewProcessName|endswith:
      - '\powershell.exe'
      - '\pwsh.exe'
      - '\cmd.exe'
  selection_powershell:
    CommandLine|contains:
      - 'powershell.exe'
      - 'pwsh.exe'
  selection_length:
    CommandLine|re: '.{1000,}'
  condition: selection and selection_powershell and selection_length
falsepositives:
  - Unknown
level: low
```

Sigma Rule Breakdown:

- `logsource`: The rule looks into logs under the category of `process_creation` and is designed to work against Windows machines.

```
logsource:
  category: process_creation
  product: windows
```

- `detection`: The `selection` section checks if any Windows events with ID `4688` exist and also checks if the `NewProcessName` field ends with `\powershell.exe`, `\pwsh.exe`, or `\cmd.exe`.
 - The `selection_powershell` section checks if the executed command line includes PowerShell-related executables and finally, the `selection_length` section checks if the `CommandLine` field of the `4688` event contains 1,000 characters or more.
 - The `condition` section checks if the selection criteria inside the `selection`, `selection_powershell`, and `selection_length` sections are all met.

```
detection:
  selection:
    EventID: 4688
```

```
NewProcessName|endswith:  
- '\powershell.exe'  
- '\pwsh.exe'  
- '\cmd.exe'  
selection_powershell:  
CommandLine|contains:  
- 'powershell.exe'  
- 'pwsh.exe'  
selection_length:  
CommandLine|re: '.{1000,}'  
condition: selection and selection_powershell and selection_length
```

- Let's put Chainsaw to work by applying the abovementioned Sigma rule,
`proc_creation_win_powershell_abnormal_commandline_size.yml` (available at
`C:\Rules\sigma`), to `lab_events_3.evtx` (available at
`C:\Events\YARASigma\lab_events_3.evtx`, thanks to `mdecrevoisier`) that contains

4688 events with abnormally long PowerShell commands.

lab_events_3 Number of events: 46

Level	Date and Time	Source	Event ID	Task Category
Information	4/22/2021 1:51:05 AM	Microsoft Wind...	4673	Sensitive Privileg...
Information	4/22/2021 1:51:04 AM	Microsoft Wind...	4688	Process Creation
Information	4/22/2021 1:51:04 AM	Microsoft Wind...	4688	Process Creation
Information	4/22/2021 1:51:04 AM	Microsoft Wind...	4688	Process Creation
Information	4/22/2021 1:51:04 AM	Microsoft Wind...	5145	Detailed File Share
Information	4/22/2021 1:51:04 AM	Microsoft Wind...	5140	File Share
Information	4/22/2021 1:51:04 AM	Microsoft Wind...	5145	Detailed File Share
Information	4/22/2021 1:51:04 AM	Microsoft Wind...	5140	File Share

Event 4688, Microsoft Windows security auditing.

General Details

A new process has been created.

Subject:

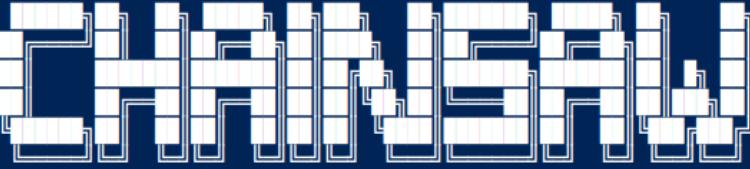
Security ID:	SYSTEM
Account Name:	FS03VULN\$
Account Domain:	OFFSEC
Logon ID:	0x3E7

Process Information:

New Process ID:	0x6e8
New Process Name:	C:\Windows\System32\cmd.exe
Token Elevation Type:	TokenElevationTypeDefault (1)
Creator Process ID:	0x1d0
Process Command Line:	C:\Windows\system32\cmd.exe /b /c start /b /min powershell.exe -nop -w hidden -noni -c "if([IntPtr]::Size -eq 4){\$b='powershell.exe'}else{\$b=\$env:windir+'\syswow64\WindowsPowerShell\v1.0\powershell.exe'};\$s=New-Object System.Diagnostics.ProcessStartInfo;\$s.FileName=\$b;\$s.Arguments='-nop -w hidden -c &([scriptblock]::create((New-Object System.IO.StreamReader(New-Object

Log Name: Security
 Source: Microsoft Windows security | Logged: 4/22/2021 1:51:04 AM
 Event ID: 4688 Task Category: Process Creation
 Level: Information Keywords: Audit Success
 User: N/A Computer: fs03vuln.offsec.lan
 OpCode: Info
 More Information: [Event Log Online Help](#)

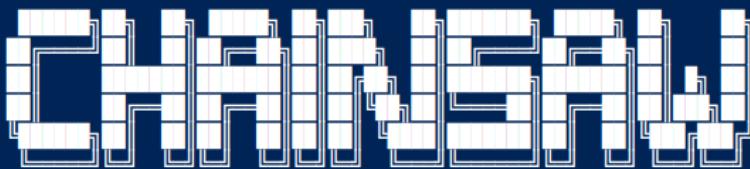
```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe hunt
C:\Events\YARASigma\lab_events_3.evtx -s
C:\Rules\sigma\proc_creation_win_powershell_abnormal_commandline_size.yml
l --mapping .\mappings\sigma-event-logs-all.yml
```

```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe hunt C:\Events\YARASigma\lab_events_3.evtx -  
  
By Countercept (@FranticTyping, @AlexKornitzer)  
[+] Loading detection rules from: C:\Rules\sigma\proc_creation_win_powershell_abnormal_commandline_size.y  
l  
[+] Loaded 1 detection rules  
[+] Loading forensic artefacts from: C:\Events\YARASigma\lab_events_3.evtx (extensions: .evt, .evtx)  
[+] Loaded 1 forensic artefacts (69.6 KB)  
[+] Hunting: [=====] 1/1 -  
[+] 0 Detections found on 0 documents
```

- Our Sigma doesn't seem to be able to identify the abnormally long PowerShell commands within these 4688 events.
- Does this mean that our Sigma rule is flawed?
 - No! As discussed previously, Chainsaw's mapping file (specified through the `--mapping` parameter) tells it which fields in the event logs to use for rule matching.
- It looks like the `NewProcessName` field was missing from the `sigma-event-logs-all.yml` mapping file.
- We introduced the `NewProcessName` field into a `sigma-event-logs-all-new.yml` mapping file inside the `C:\Tools\chainsaw\mappings` directory of this section's target.
- Let's run Chainsaw again with this new mapping file.

```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe hunt  
C:\Events\YARASigma\lab_events_3.evtx -s  
C:\Rules\sigma\proc_creation_win_powershell_abnormal_commandline_size.y  
l --mapping .\mappings\sigma-event-logs-all-new.yml
```

```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe hunt C:\Events\YARASigma\lab_events_3.evtx -
```



By Countercept (@FranticTyping, @AlexKornitzer)

```
[+] Loading detection rules from: C:\Rules\sigma\proc_creation_win_powershell_abnormal_commandline_size.  
[+] Loaded 1 detection rules  
[+] Loading forensic artefacts from: C:\Events\YARASigma\lab_events_3.evtx (extensions: .evtx, .evt)  
[+] Loaded 1 forensic artefacts (69.6 KB)  
[+] Hunting: [=====] 1/1 -  
[+] Group: Sigma
```

timestamp	detections	count	Event.System.Provider	Event ID
2021-04-22 08:51:04	+ Unusually Long PowerShell CommandLine	1	Microsoft-Windows-Security-Auditing	4688

- Our Sigma rule successfully uncovered all three abnormally long PowerShell commands that exist inside `lab_events_3.evtx`
- Remember that configuration when it comes to using or translating Sigma rules is of paramount importance!

One sentence summary

- Chainsaw with appropriate mapping file can act as an SIEM for threat hunting purposes.

Hunting Evil with Sigma (Splunk Edition)

Summary

- As discussed when introducing Sigma, Sigma rules revolutionize our approach to log analysis and threat detection.
 - What we're dealing with here is a sort of Rosetta Stone for SIEM systems.
 - Sigma is like a universal translator that brings in a level of abstraction to event logs, taking away the painful element of SIEM-specific query languages.
- Let's validate this assertion by converting two Sigma rules into their corresponding SPL formats and examining the outcomes.

Example 1: Hunting for MiniDump Function Abuse to Dump LSASS's Memory (comsvcs.dll via rundll32)

- A Sigma rule named `proc_access_win_lsass_dump_comsvcs_dll.yml` can be found inside the `C:\Tools\chainsaw\sigma\rules\windows\process_access` directory of the previous section's target.
- This Sigma rule detects adversaries leveraging the `MiniDump` export function of `comsvcs.dll` via `rundll32` to perform a memory dump from LSASS.
- We can translate this rule into a Splunk search with `sigmac` (available at `C:\Tools\sigma-0.21\tools`) as follows.

```
PS C:\Tools\sigma-0.21\tools> python sigmac -t splunk
C:\Tools\chainsaw\sigma\rules\windows\process_access\proc_access_win_lsass_dump_comsvcs_dll.yml -c .\config\splunk-windows.yml

(TargetImage="*\lsass.exe"
SourceImage="C:\Windows\System32\rundll32.exe"
CallTrace="*comsvcs.dll")
```

The screenshot shows the Splunk Enterprise search interface. The search bar contains the command: `targetImage="*\lsass.exe" SourceImage="C:\Windows\System32\rundll32.exe" CallTrace="*comsvcs.dll"`. The results pane shows 3 events from August 8, 2022, at 11:46:07 AM. The first event is expanded, showing details about the logon process and the call trace. The event ID is 1103, and the timestamp is 11/8/2022 11:46:07 AM. The event type is 4 (Informational). The computer name is DESKTOP-EG551S. The call trace path is `C:\Windows\SYSTEM32\ntdll.dll!9d4c4C\Windows\SYSTEM32\ntdll.dll!d7ca1C\Windows\System32\KERNEL32.DLL!1dec1C\Windows\System32\KERNEL32.DLL+265e5C\Windows\SYSTEM32\rbcore.DLL+99b1C\Windows\SYSTEM32\rbcore.DLL+179b5C\Windows\SYSTEM32\rbcore.DLL+6222C\Windows\SYSTEM32\rbcore.DLL+6cfb1C\Windows\System32\comsvcs.dll+2202C\Windows\system32\rundll32.exe+42eb1C\Windows\sytem32\rundll32.exe+679eC\Windows\System32\KERNEL32.DLL+17034C\Windows\SYSTEM32\ntdll.dll+526a1`. The event source is WinEventLog-Sysmon, and the log name is Microsoft-Windows-Sysmon/Operational.

- The Splunk search provided by `sigmac` was indeed able to detect MiniDump function abuse to dump LSASS's memory.

Example 2: Hunting for Notepad Spawning Suspicious Child Process

- A Sigma rule named `proc_creation_win_notepad_susp_child.yml` can be found inside the `C:\Rules\sigma` directory of the previous section's target.
- This Sigma rule detects `notepad.exe` spawning a suspicious child process.
- We can translate this rule into a Splunk search with `sigmac` (available at `C:\Tools\sigma-0.21\tools`) as follows.

```
PS C:\Tools\sigma-0.21\tools> python sigmac -t splunk
C:\Rules\sigma\proc_creation_win_notepad_susp_child.yml -c
.\config\splunk-windows.yml

(ParentImage="*\\"notepad.exe" (Image="*\\"powershell.exe" OR
Image="*\\"pwsh.exe" OR Image="*\\"cmd.exe" OR Image="*\\"mshta.exe" OR
Image="*\\"cscript.exe" OR Image="*\\"wscript.exe" OR
Image="*\\"taskkill.exe" OR Image="*\\"regsvr32.exe" OR
Image="*\\"rundll32.exe" OR Image="*\\"calc.exe" ))
```

The screenshot shows a Splunk search interface. The search bar contains the query: 1 [!parentImage=*\\notepad.exe" (Image=*\\powershell.exe OR Image=*\\pwsh.exe OR Image=*\\cmd.exe OR Image=*\\mshta.exe OR Image=*\\vbscript.exe OR Image=*\\wscript.exe OR Image=*\\taskkill.exe OR Image=*\\regsvr32.exe" OR Image=*\\rundll32.exe" OR Image=*\\calc.exe")]. The results table shows 21 events. One event is selected, showing details like LogName=Microsoft-Windows-Sysmon/Operational, EventCode=1, EventType=4, ComputerName=DESKTOP-EGSS5IS.uniwaldo.local, and a long JSON payload describing a PowerShell process creation.

- The Splunk search provided by `sigmac` was indeed able to detect `notepad.exe` spawning suspicious processes (such as PowerShell).

One sentence summary

- Sigmac act as an universal translator for SIEM queries and it can work for splunk as well!