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.

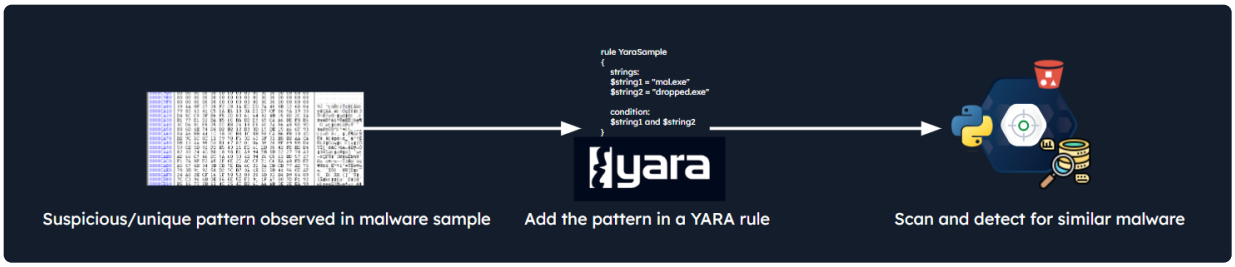
Leveraging YARA

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.

```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
  - f 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
```

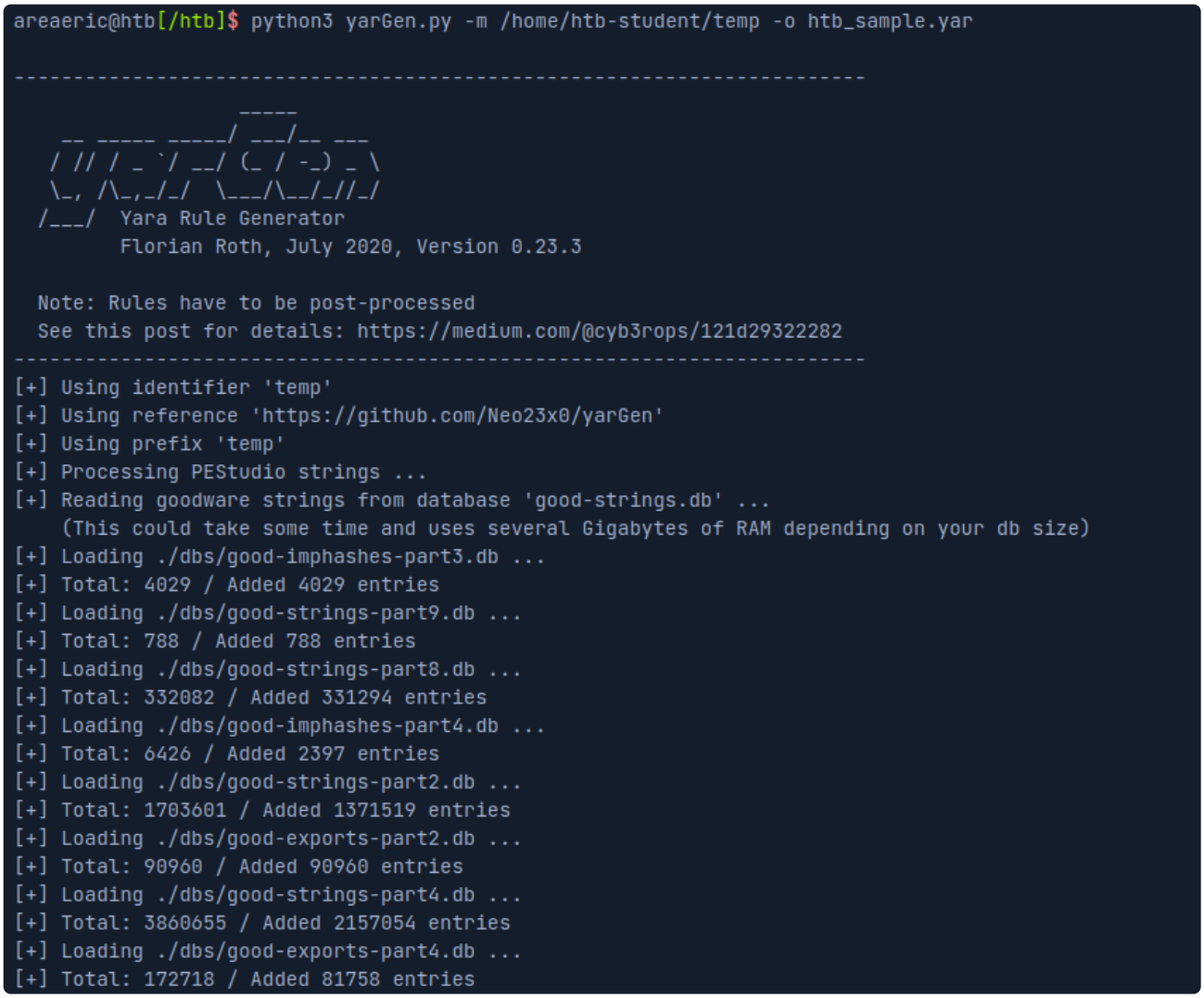
Developing YARA Rules

```
areaeric@htb[/htb]$ strings dharma_sample.exe
!This program cannot be run in DOS mode.
Rich
.text
`.rdata
@.data
9A s
---SNIP---
~?h@
~?hP
hz-A
u       jd
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@>@@@?456789:;<=@@@@@@@
@@@@@@
 !"#$%&'()*+,-./0123@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
---SNIP---
GetProcAddress
LoadLibraryA
WaitForSingleObject
InitializeCriticalSectionAndSpinCount
LeaveCriticalSection
GetLastError
EnterCriticalSection
ReleaseMutex
CloseHandle
KERNEL32.dll
RSDS%~m
C:\crysis\Release\PDB\payload.pdb
---SNIP---
```

- 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
```



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 = "bff6a1000a86f8edf3673d576786ec75b80bed0c458a8ca0bd52d12b74099071"
   strings:
      $x1 = "C:\\crysis\\Release\\PDB\\payload.pdb" fullword ascii
      $s2 = "sssssbs" fullword ascii
      $s3 = "sssssbsss" 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
```

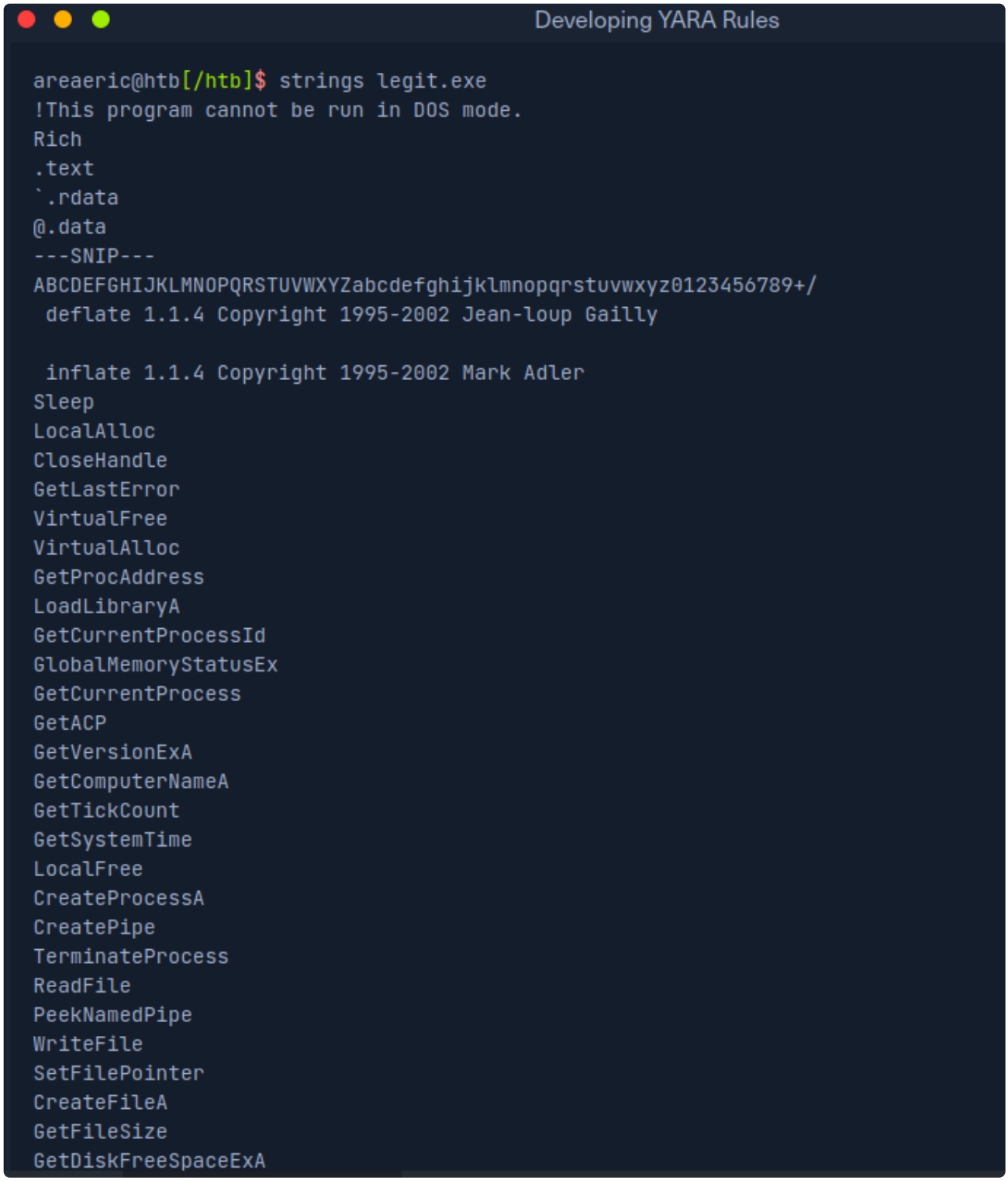
- As we can see, the `pdf_reader.exe`, `microsoft.com`, `check_updates.exe`, and `KB5027505.exe` files are detected by this rule (in addition to `dharma_sample.exe` of course).

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.

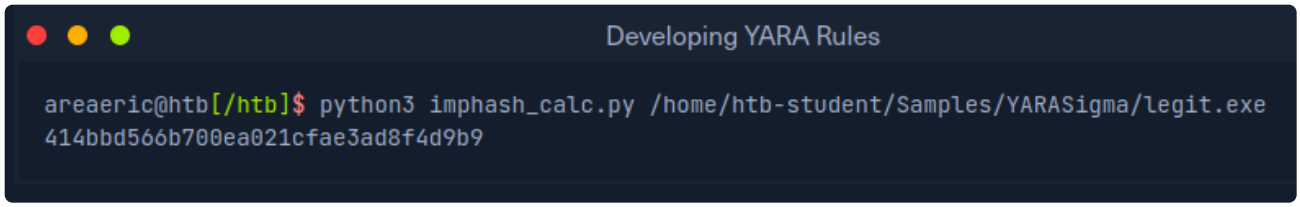
```
areaeric@htb[/htb]$ strings legit.exe
```



- 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
```



- 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&tbm=isus&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() == "414bbd566b700ea021cfae3ad8f4d9b9"`: 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
```

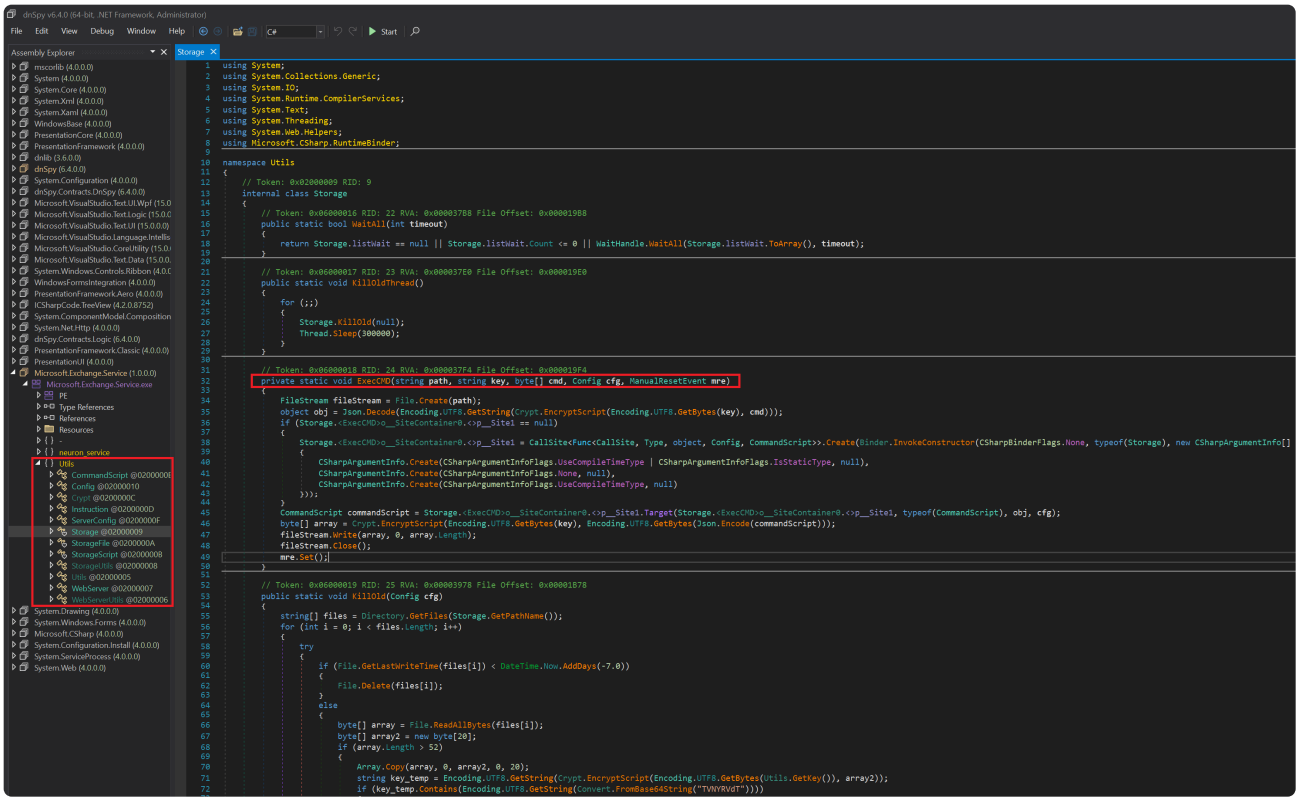
```
areaeric@htb[/htb]$ cat code
```

```
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.Co
---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 manage
    } // end of method Crypt::EncryptScript
            IL_00a0:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned in
          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.



- 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 = "FindSPath" ascii
    $dotnetMagic = "BSJB" ascii
  condition:
    (uint16(0) == 0x5A4D and uint16(uint32(0x3c)) == 0x4550) and
$dotnetMagic and 6 of them
}
```

YARA Rule Breakdown:

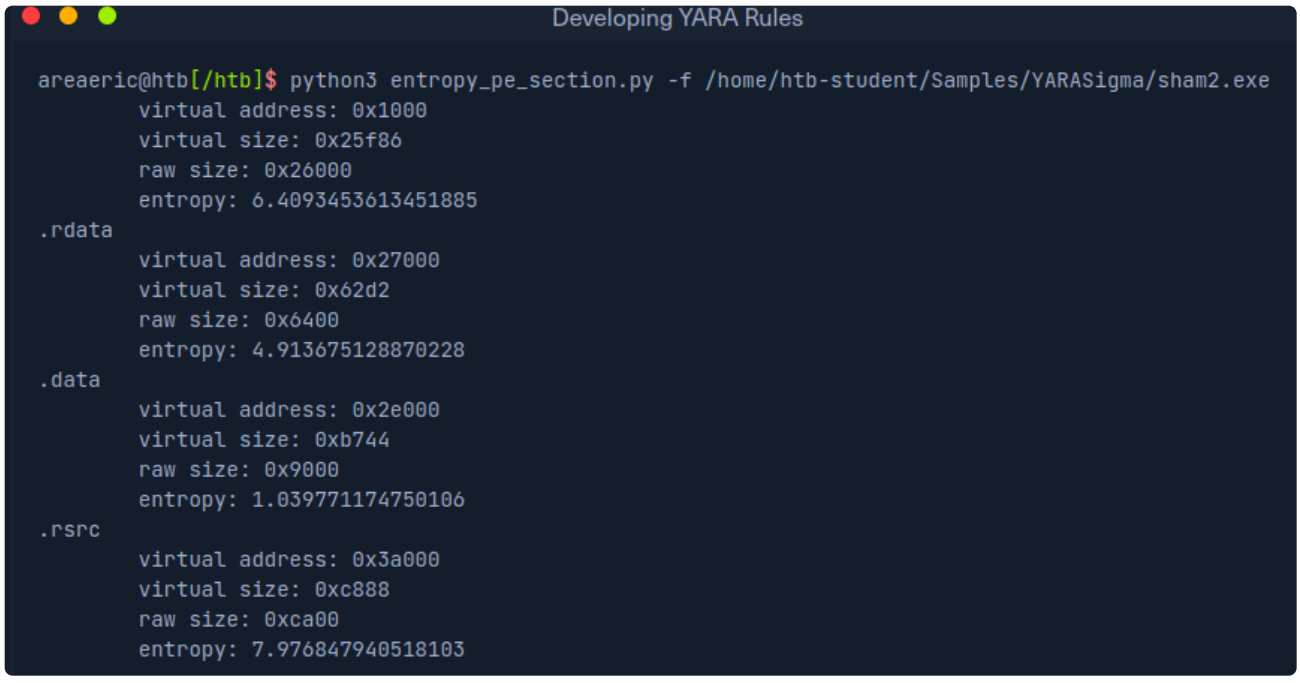
- **`Strings Section`**:
  - `$class1 = "StorageUtils" ascii` to `$class8 = "W3WPDIAG" ascii`: These are eight ASCII strings corresponding to class names within the .NET assembly.
  - `$func1 = "AddConfigAsString" ascii` to `$func7 = "FindSPath" ascii`: These seven ASCII strings represent class or function 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
```



- 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", $a2 = "FindNextFile"`: Strings for Windows API functions used to enumerate files.
    - The usage of `FindFirstFileW` and `FindNextFileW` API functions can be idenfitied through string analysis.
  - `$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

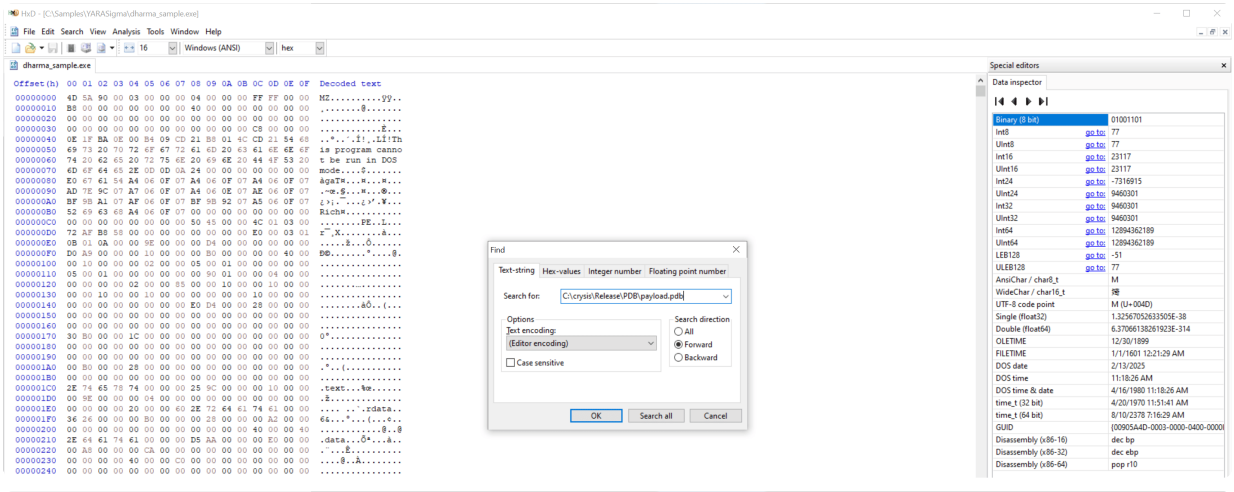
- As you can imagine, the best YARA rule development resource is the official documentation, which can be found at the following link.
- The next best resource on effective YARA rule developement comes from Kaspersky.
- Below are some blog posts that offer a more detailed explanation on how to use `yarGen` 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
- `yarGen` is a great tool to develop some good yara rules by extracting unique patterns.
    - Once the rule is developed with the help of yarGen, 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 yarGen 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.

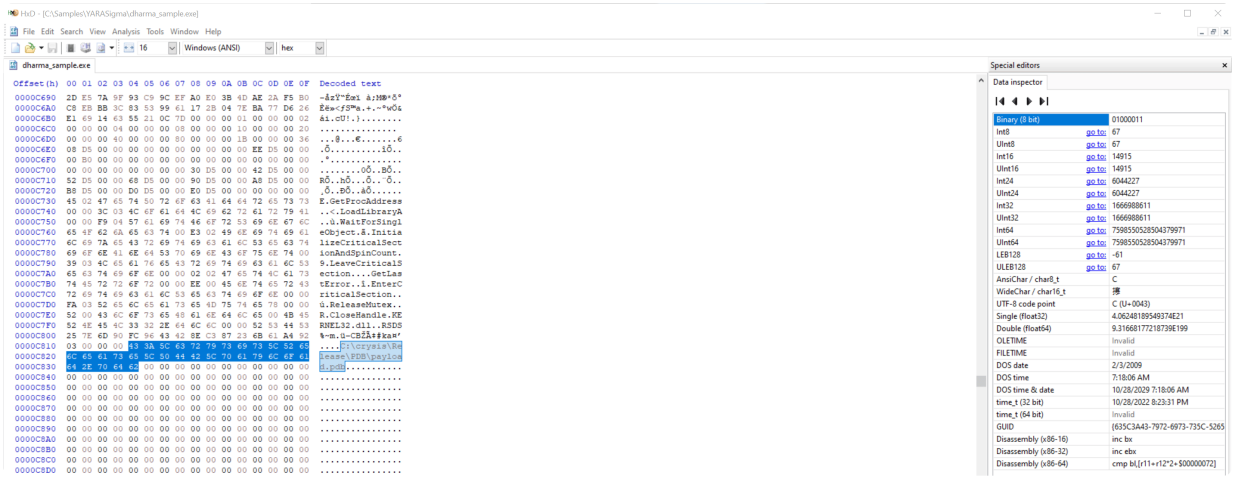
One sentence summary

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

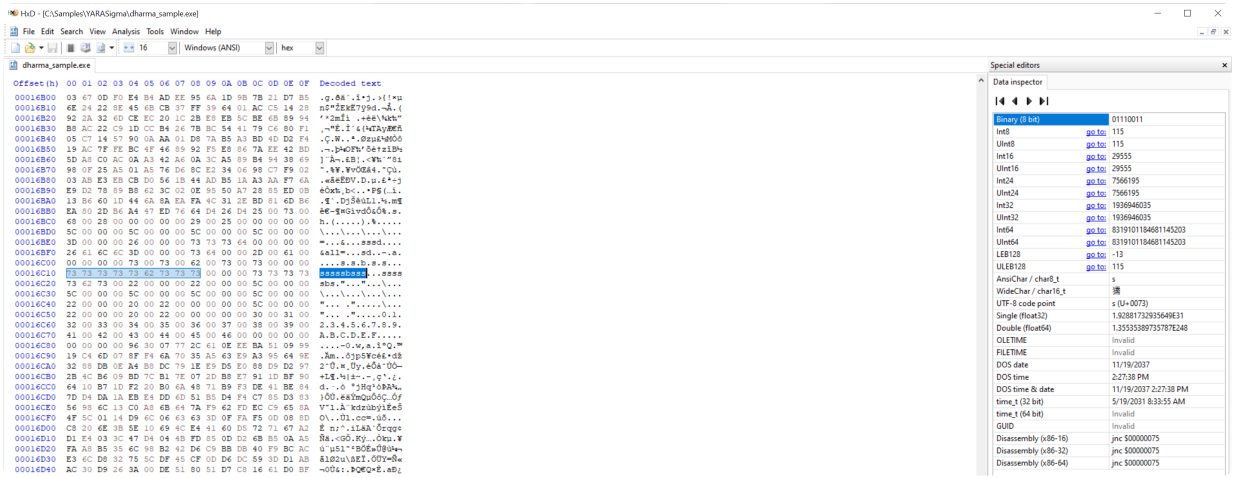
Hunting Evil with YARA (Windows Edition)

- 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.

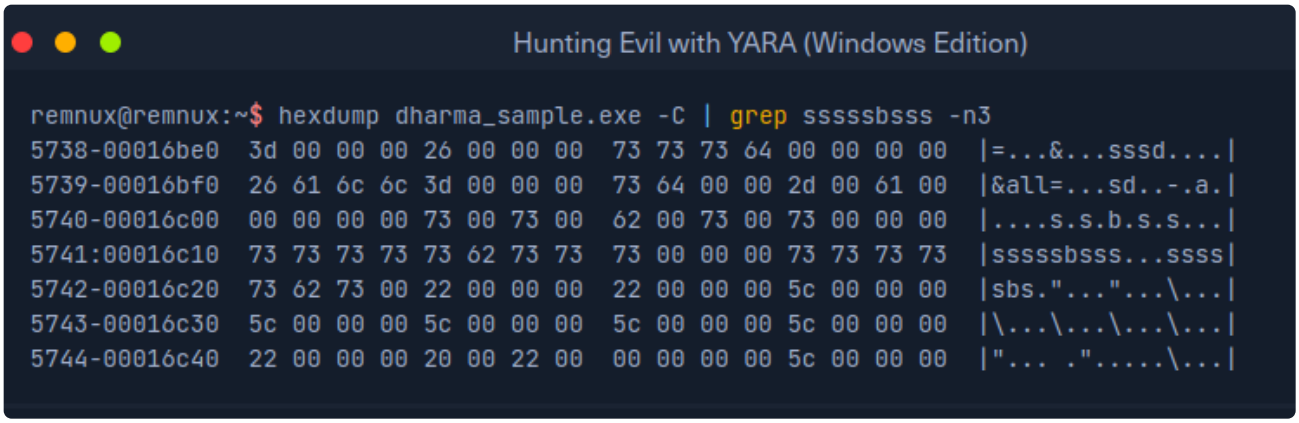


- Going forward, we will craft a rule grounded in these patterns and then utilize the YARA utility to scour the filesystem for similar executables.

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

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


```
remnux@remnux:~$ hexdump dharma_sample.exe -C | grep sssssbsss -n3
```



- 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
```

- **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.

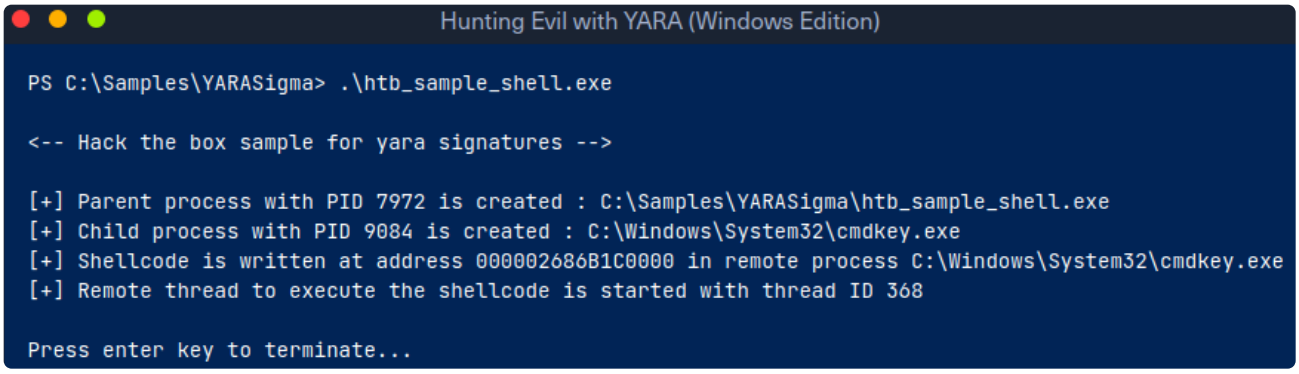
YARA Rule Source:
https://github.com/cuckoosandbox/community/blob/master/data/yara/shellcode/metasploit.yar

```
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 prologe in
metasploit
        $s2 = { 648b ??30 }             // mov edx, fs:[???+0x30]
        $s3 = { 4c77 2607 }             // kernel32 checksum
        $s4 = "ws2_"                    // ws2_32.dll
        $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 targe
- `htb_sample_shell.exe` injects Metasploit's meterpreter shellcode into the `cmdkey.exe` process. Let's activate it, ensuring successful injection.

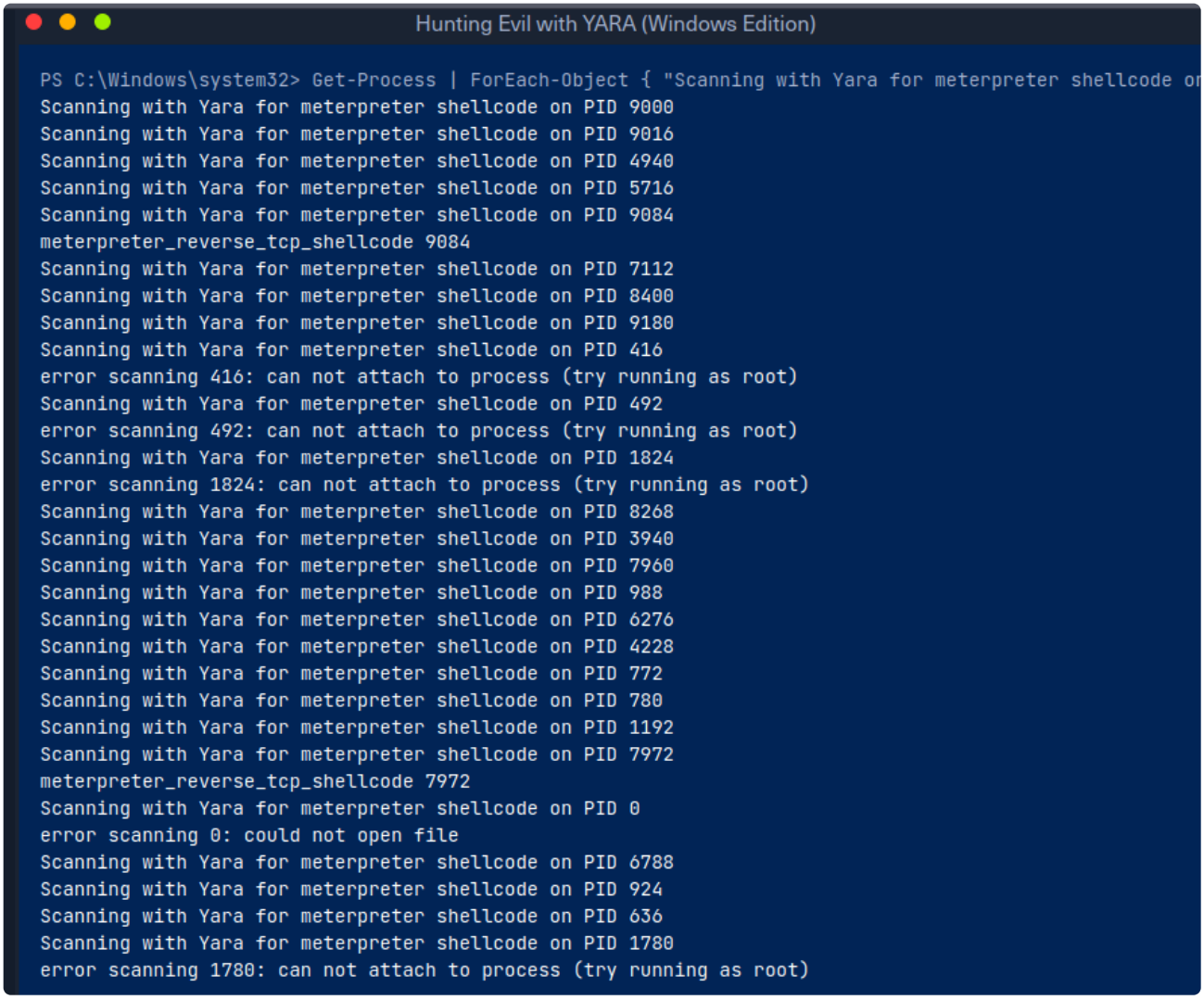
Note: Make sure you launch PowerShell as an administrator.

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



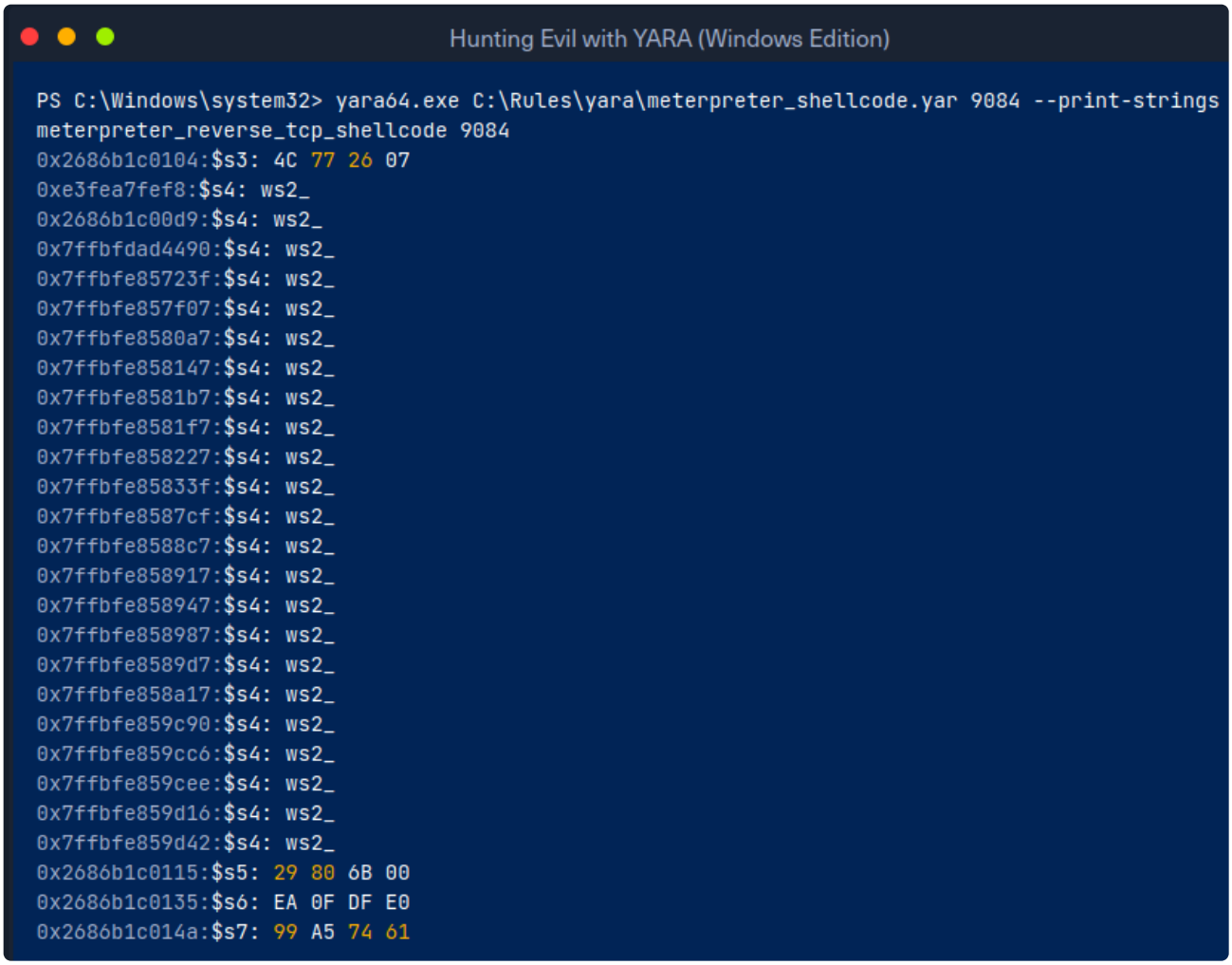
- With the injection executed, let's scan every active system process as follows, through another PowerShell terminal (`Run as administrator`).

```
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 }
```

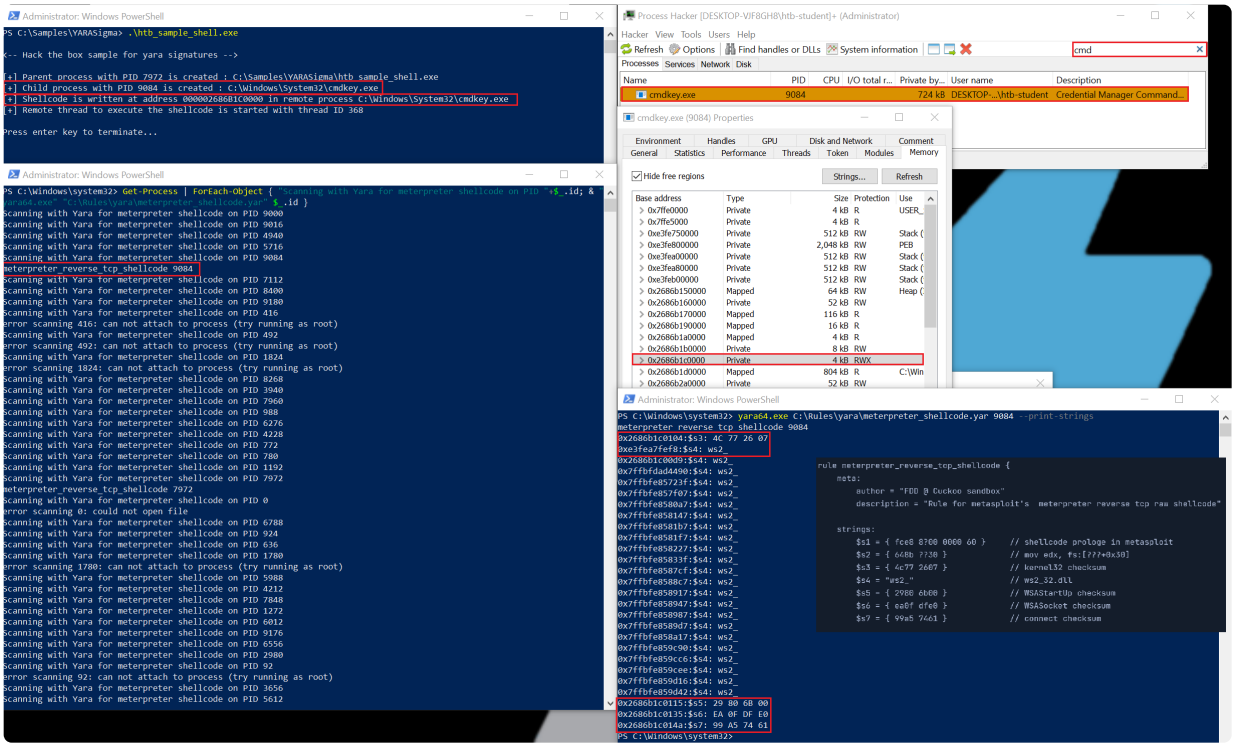


- We're leveraging a concise PowerShell script.
  - The `Get-Process` command fetches running processes, and with the help of the pipe symbol (`|`), this data funnels into the script block (`{...}`).
  - Here, `ForEach-Object` dissects each process, prompting `yara64.exe` to apply our YARA rule on each 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
```



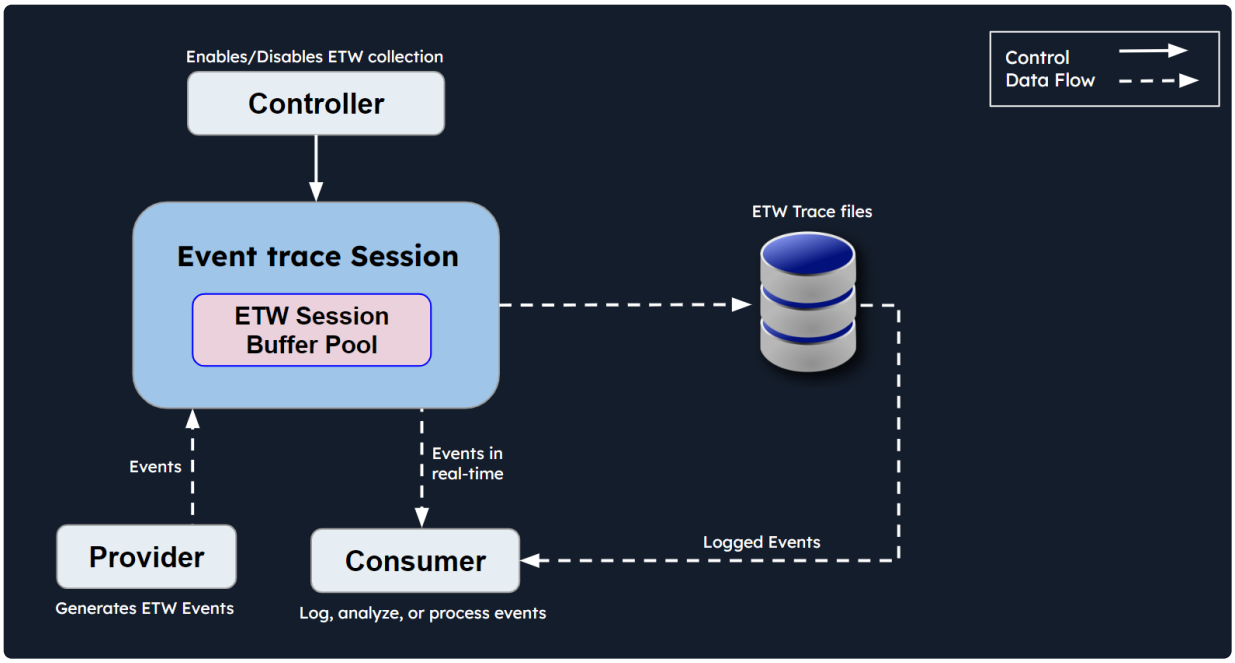
- The screenshot below shows an overview of what we discussed above.



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.

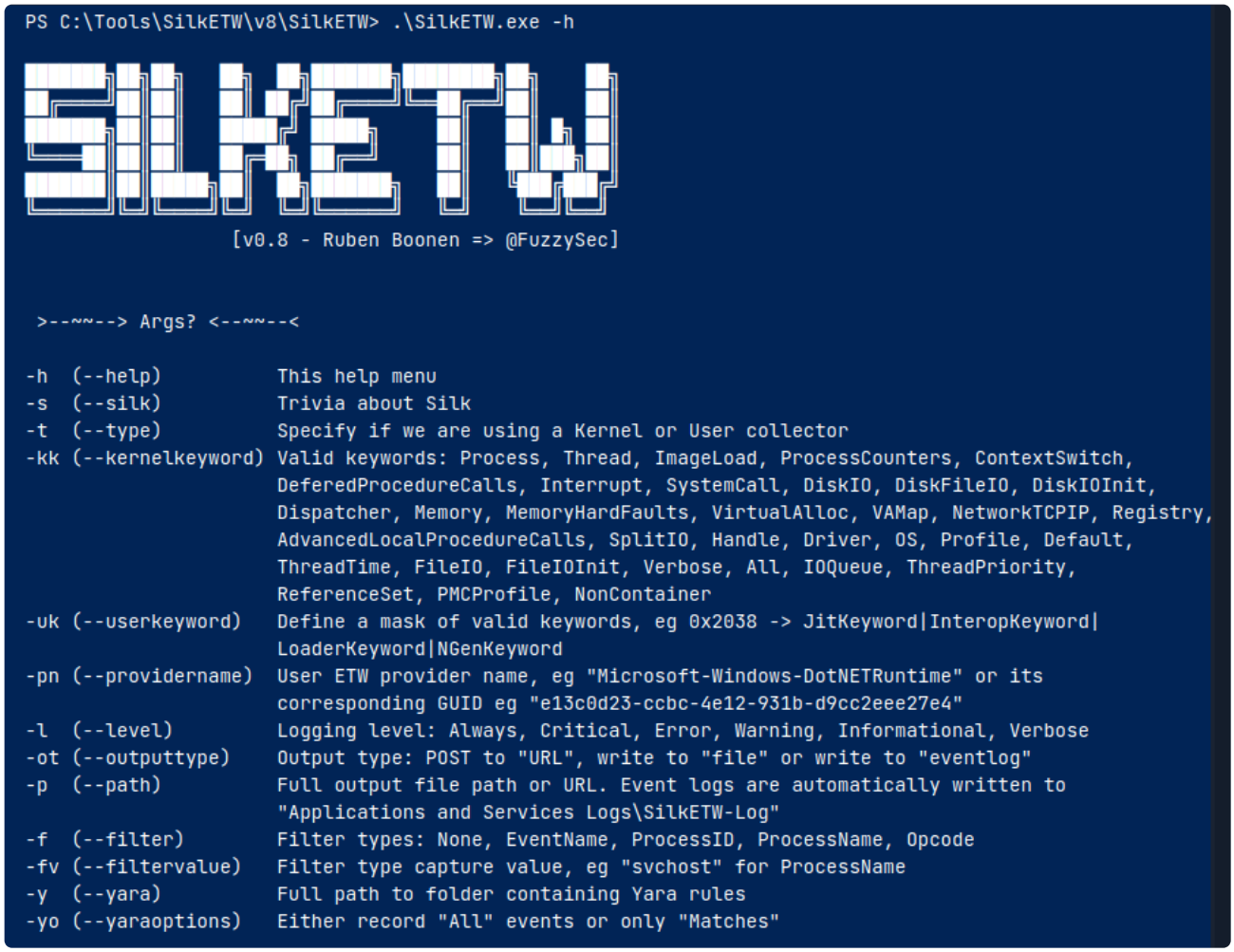


- `Controllers`: Controllers possess functionalities that encompass initiating and terminating trace sessions.
  - They also have the capability to enable or disable providers within a specific trace.
- `Providers`: Providers are crucial, as they generate events and channel them to the designated ETW 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
```



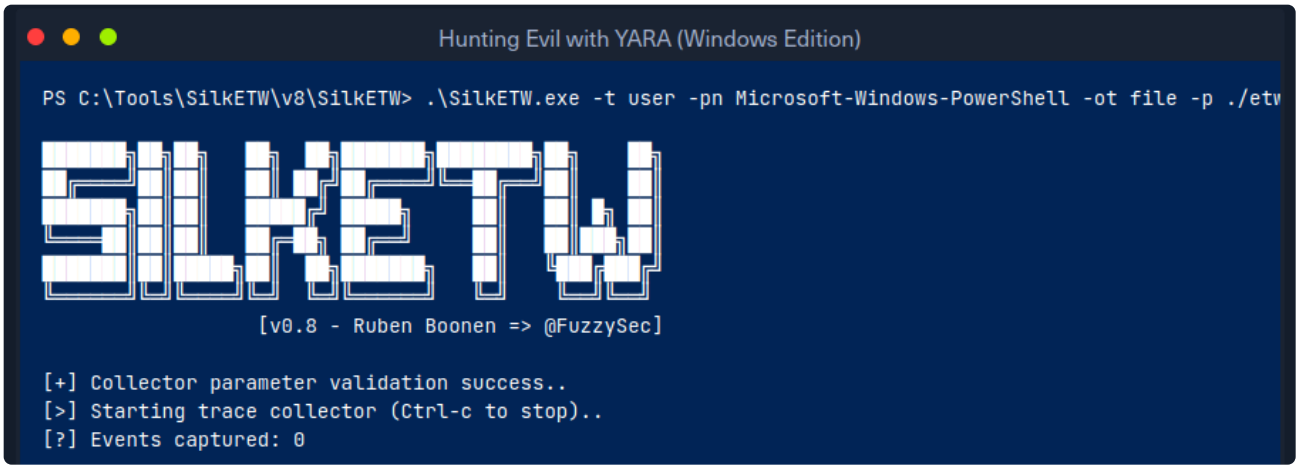
- The help menu provides many examples of how we can use the tool. Let's experiment with some of the YARA scanning options on a few ETW providers.

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
```



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.

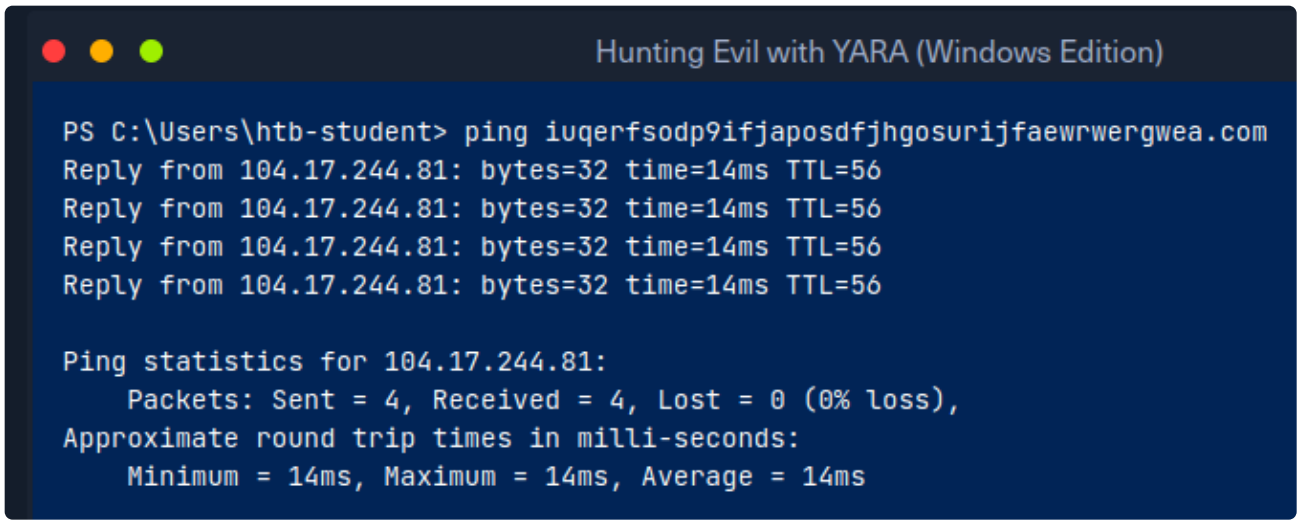
```
PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-
Windows-DNS-Client -ot file -p ./etw_dns_logs.json -l verbose -y
C:\Rules\yara  -yo Matches
```

- 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 = "iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com"
ascii wide nocase
        condition:
                $s1
}
```

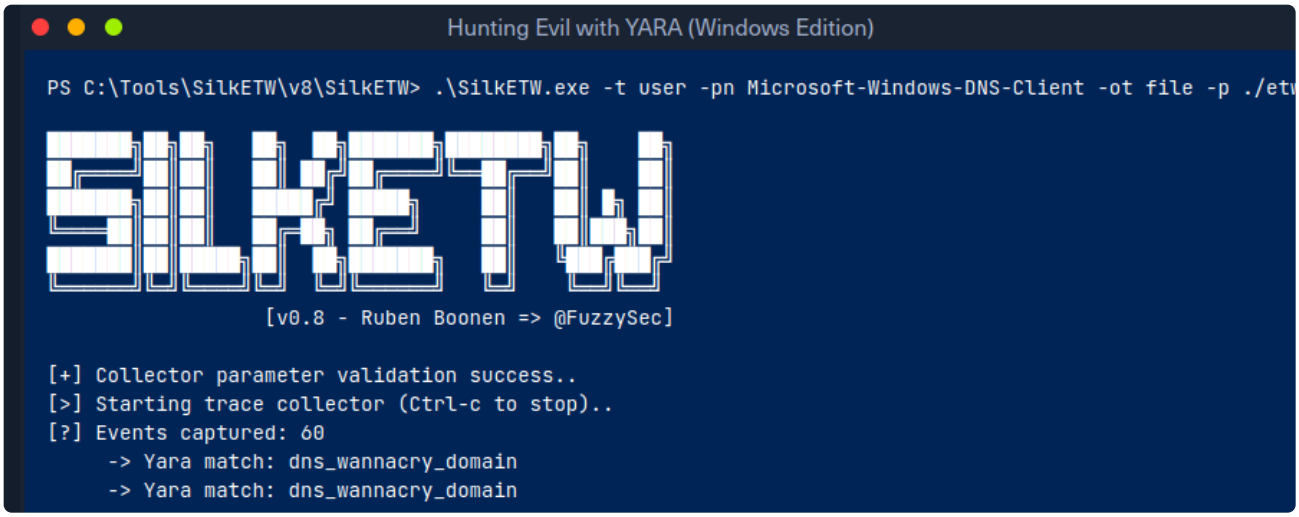
- Let's now execute the following command through another PowerShell terminal and see if it will get detected by SilkETW (where the abovementioned YARA rule has been loaded).

```
PS C:\Users\htb-student> ping
iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
```



- 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
```



One sentence summary

- The process of hunting with YARA is that need to first analyse the bahviour of the software then identify its unique behaviour to put it in the YARA for ideitifiction.
    - In WIndows we have hex editor, silketw 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 DumpIt, 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/Me
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:$wannacry_payload_str2: www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
0x197ba5e0:$wannacry_payload_str2: www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
0x1a07cedf:$wannacry_payload_str2: www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
0x1a2cb300:$wannacry_payload_str2: www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
0x1b644cd8:$wannacry_payload_str2: www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
0x1d15945b:$wannacry_payload_str2: www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
0x1dd65c9f:$wannacry_payload_str2: www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
0x450b048:$wannacry_payload_str3: mssecsvc.exe
0x5a7f3d4:$wannacry_payload_str3: mssecsvc.exe
0xda1c350:$wannacry_payload_str3: mssecsvc.exe
0x12481048:$wannacry_payload_str3: mssecsvc.exe
0x17027910:$wannacry_payload_str3: mssecsvc.exe
0x17f0dc18:$wannacry_payload_str3: mssecsvc.exe
```

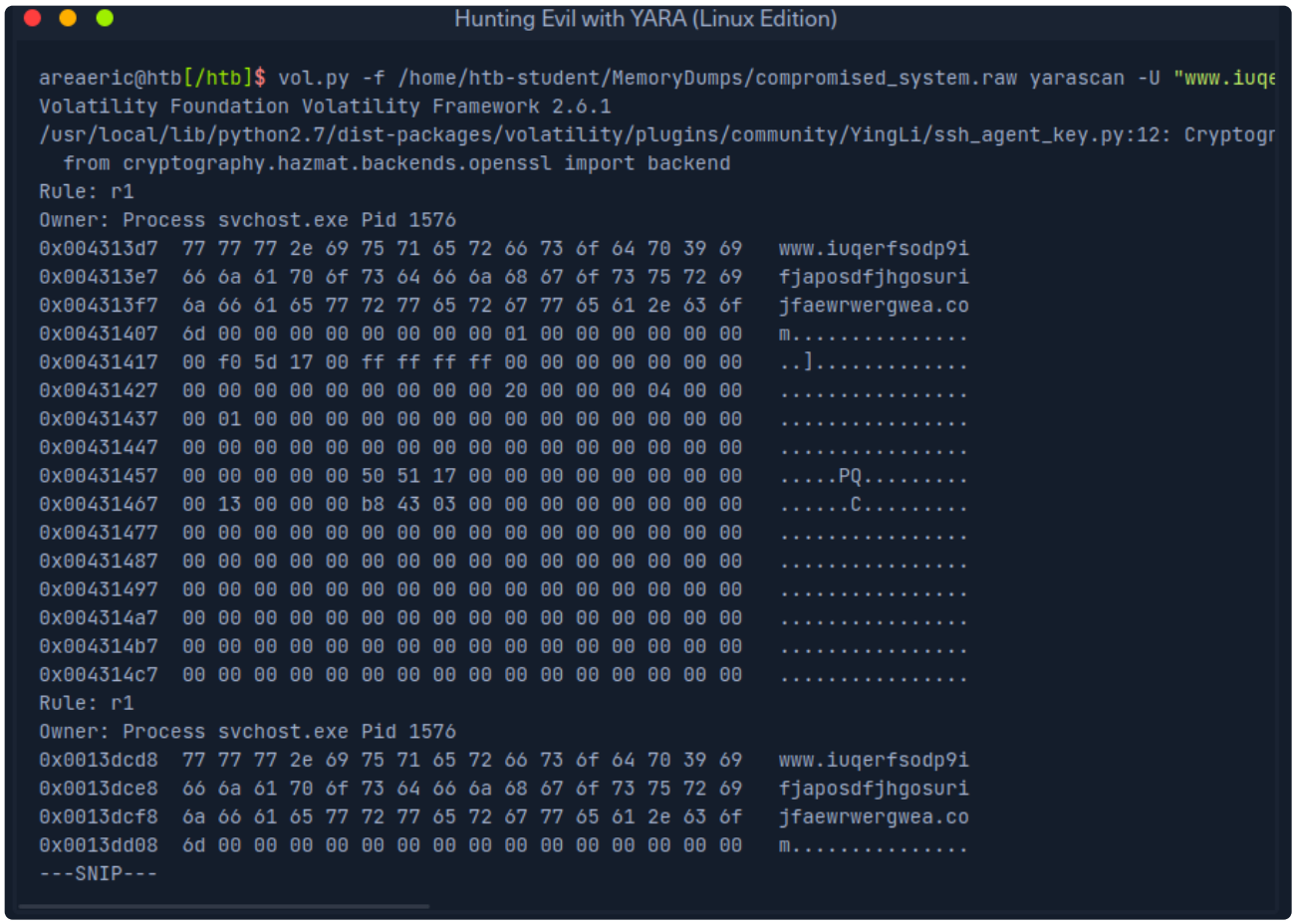
- Beyond standalone tools, diving deeper into memory forensics offers a plethora of avenues.
    - Integrating YARA within memory forensics frameworks amplifies its potential.
    - With the Volatility framework and YARA operating in tandem, WannaCry-specific IOCs can be detected seamlessly.

- 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.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com`
- Introducing this pattern within the command line using `-U "www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.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.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com"
```

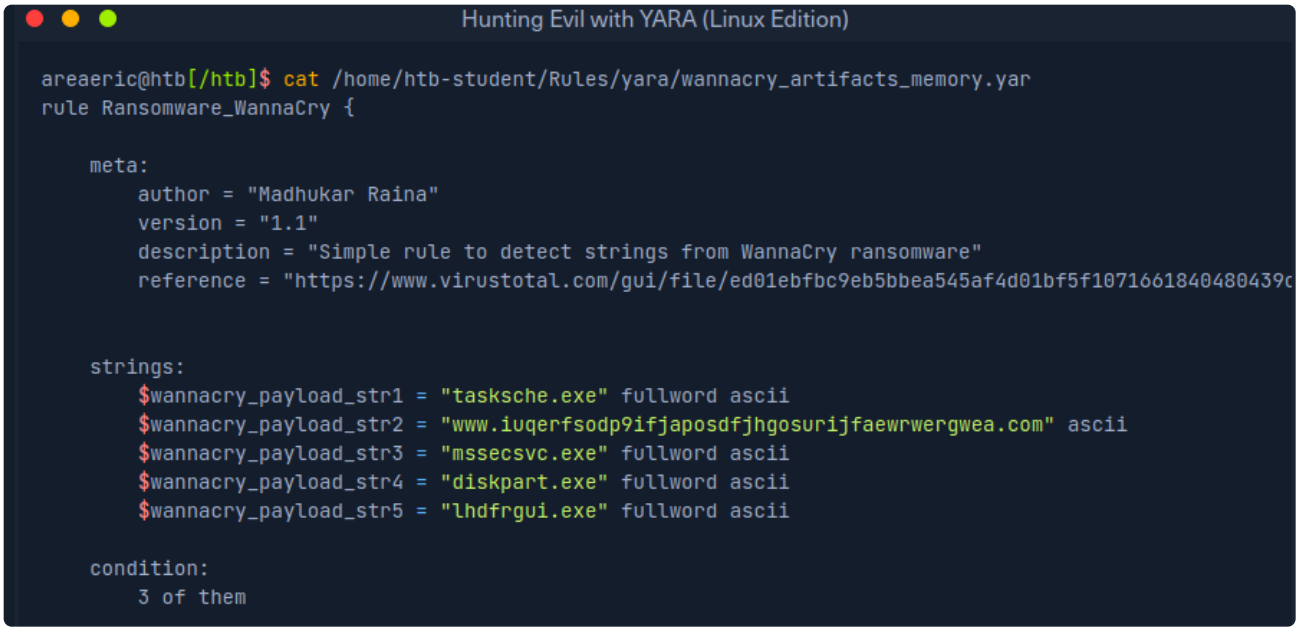


- This option allows us to directly specify a YARA rule string within the command-line itself.
  - Let's see how we can search for the content of a whole YARA rule file (i.e. `.yar` rule file) in memory image files.

Multiple YARA Rule Scanning Against a Memory Image

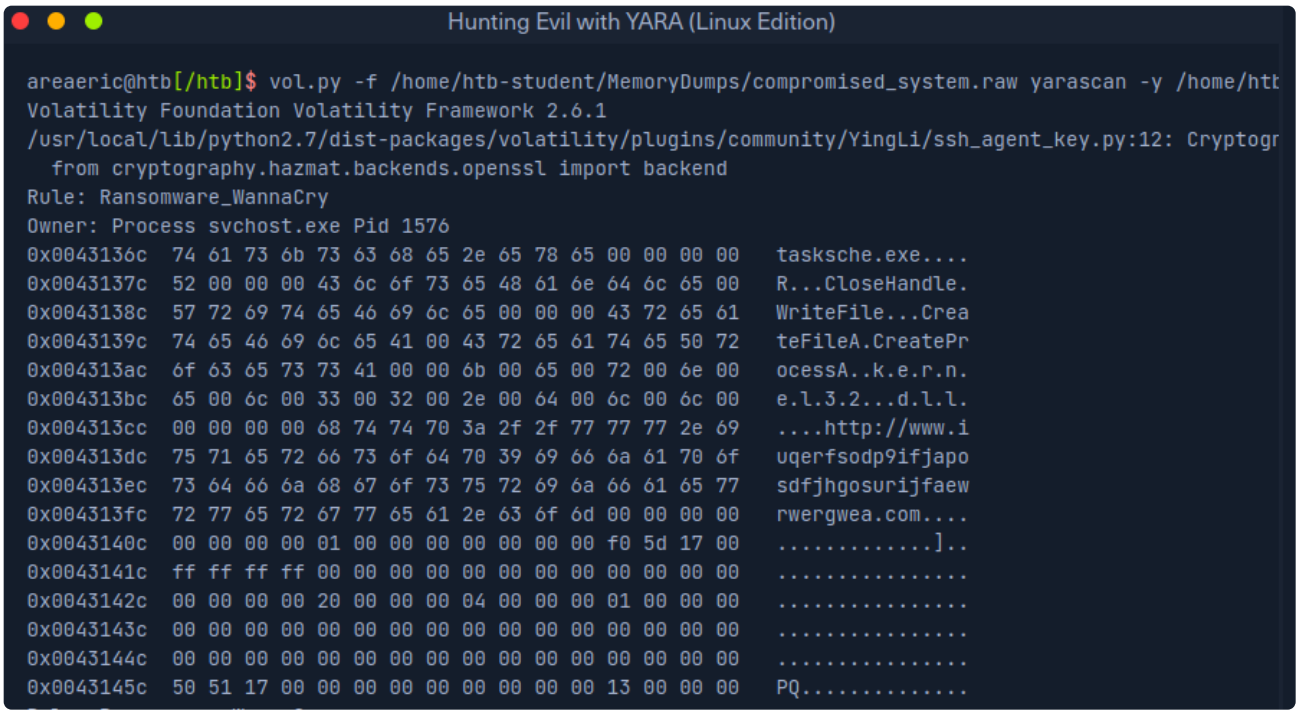
- When we have multiple YARA rules or a set of complex rules that we want to apply to a memory image, we can use the `-y` option followed by the rule file path in the Volatility framework, which allows us to specify the path to a YARA rules file.
  - The YARA rules file (`wannacry_artifacts_memory.yar`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

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



- 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
```



- 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/forsencis of the compromised server, so volatile with yara becomes a good tool for this type of work.

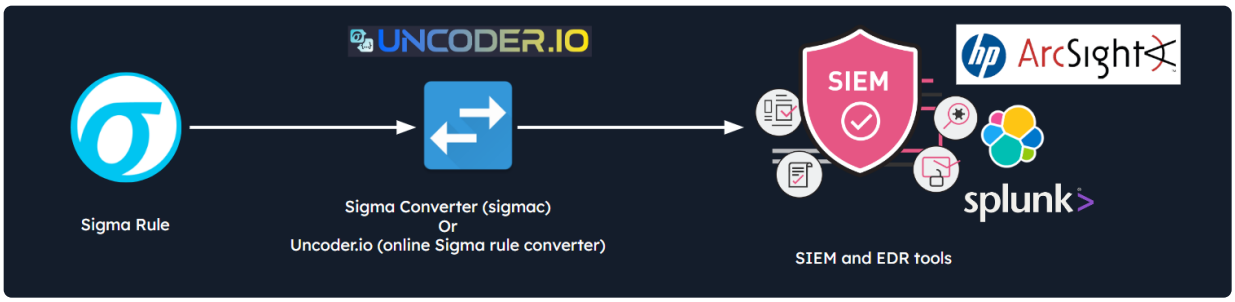
Leveraging Sigma

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.

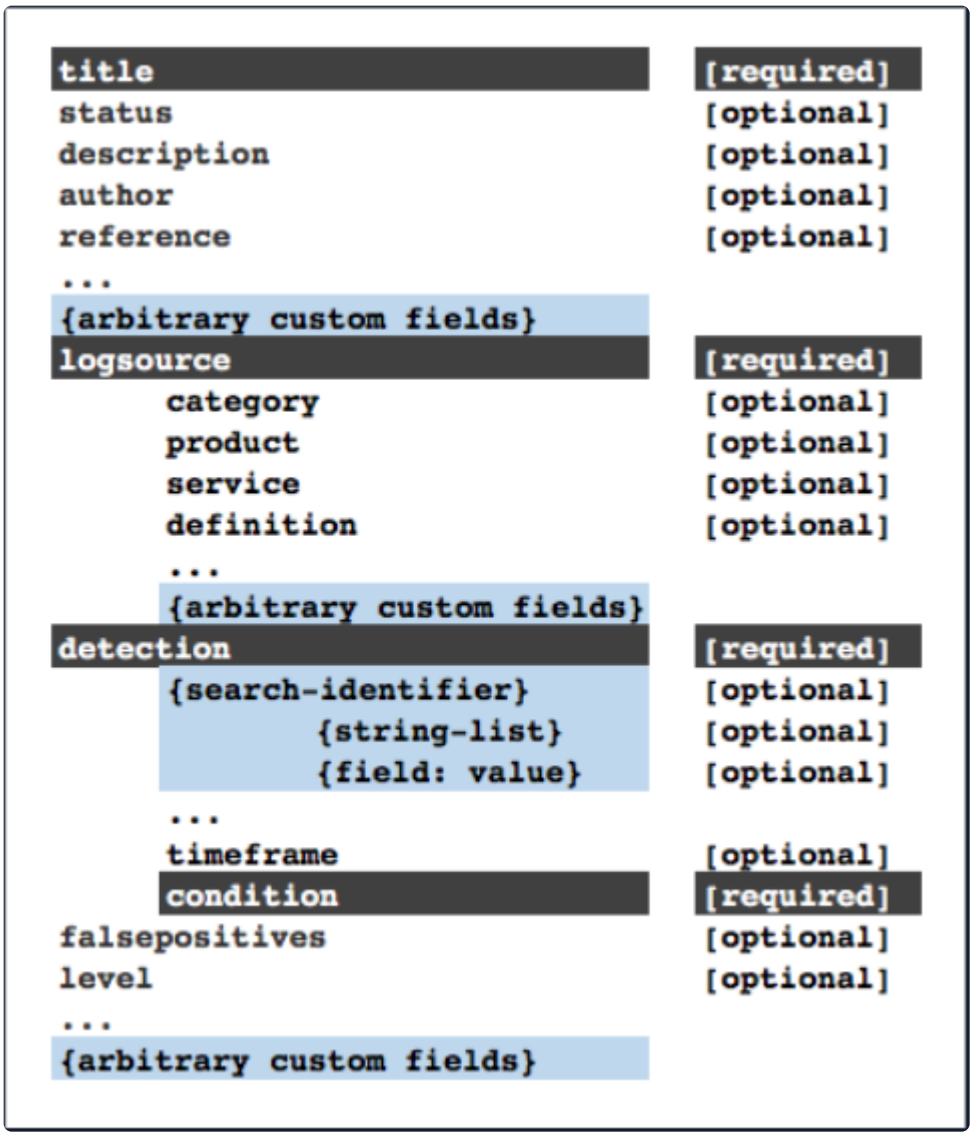
How Does Sigma Work?

- 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.



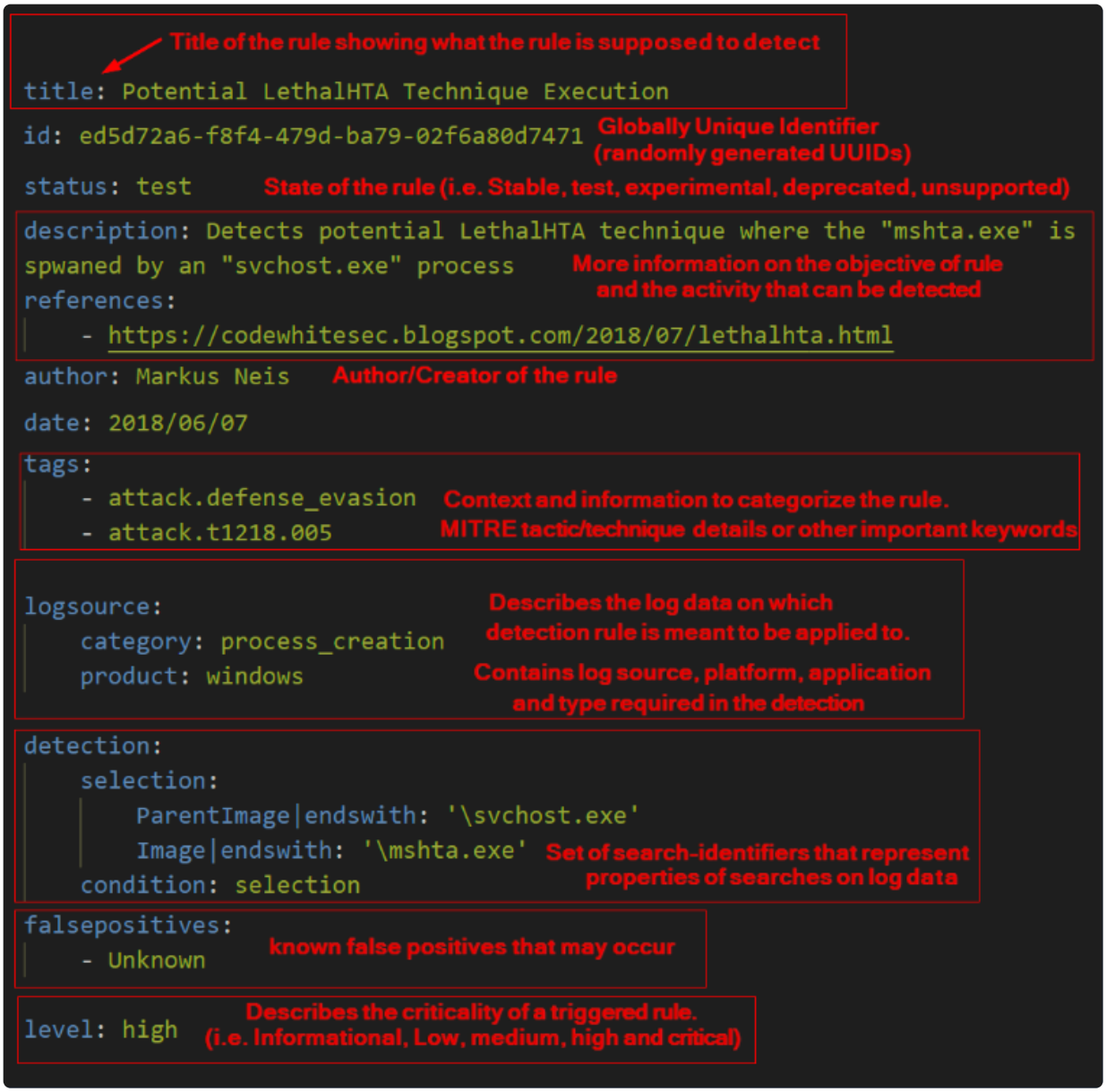
Source: https://github.com/SigmaHQ/sigma/wiki/Specification

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

Code: yaml

```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:
    - 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:



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)

```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.

```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.)

```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)

```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.

```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:
    - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
...
```

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

```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:
    - https://codewhitesec.blogspot.com/2018/07/lethalhta.html
author: Markus Neis
...
```

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

```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:
    - 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.
```

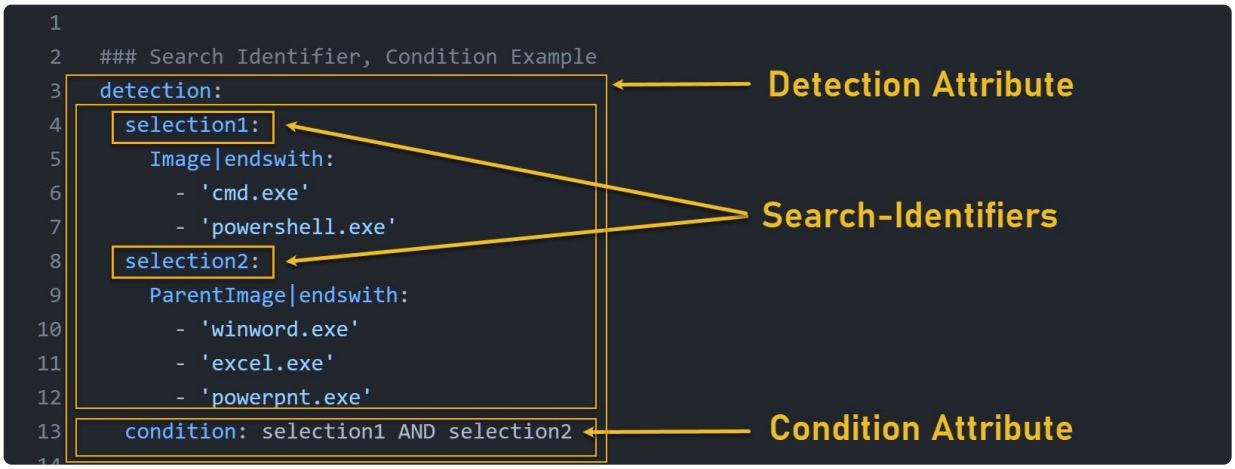
```
- `product`: The `product` value is used to select all log outputs of a certain product, e.g. all Windows event log types including `Security`, `System`, `Application` and newer types like `AppLocker` and `Windows Defender`. Examples: `windows`, `apache`, `check point fw1`, etc.

- `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

```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:
    - 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
```



Code: yaml

```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:
    - 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 :

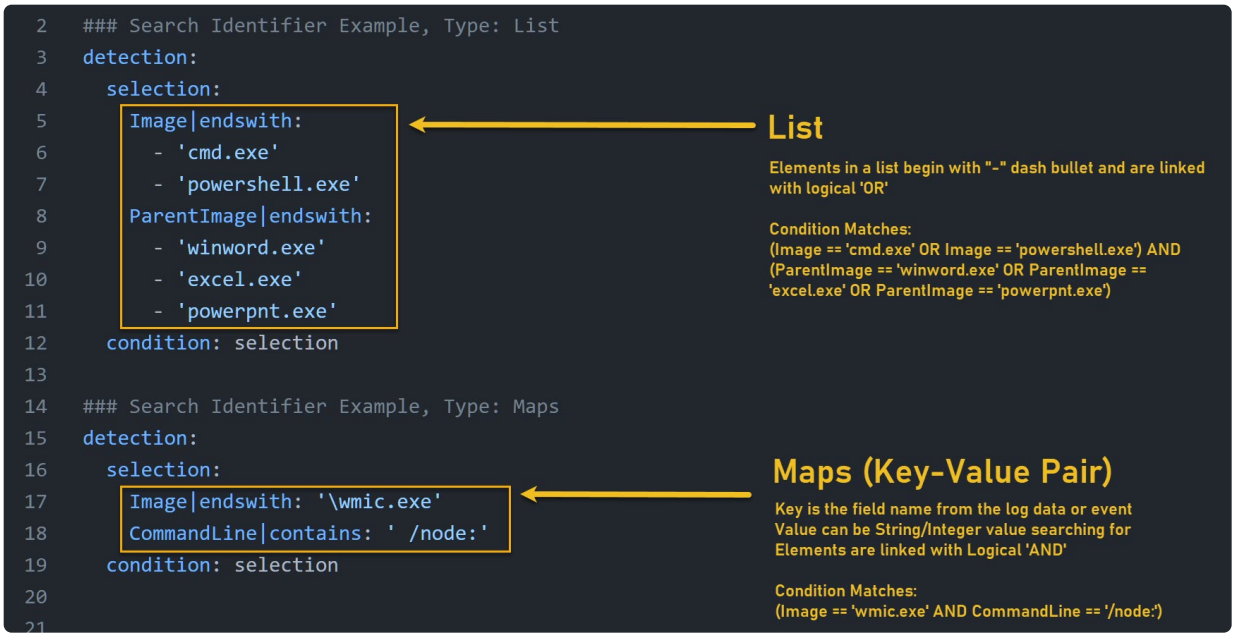
Value Modifier	Explanation	Example
contains	Adds wildcard (*) characters around the value(s)	CommandLine|contains
all	Links all elements of a list with a logical "AND" (instead of the default "OR")	CommandLine|contains|all
startswith	Adds a wildcard (*) character at the end of the field value	ParentImage|startswith
endswith	Adds a wildcard (*) character at the begining 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. **Source**: blusapphire.io

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

```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`.

```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`)

```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`.

```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	`keywords1 or keywords2`
1/all of them	`all of them`
1/all of search-identifier-pattern	`all of selection*`
1/all of search-id-pattern	`all of filter_*`
Negation with 'not'	`keywords and not filters`
Brackets - Order of operation '()'	`selection1 and (keywords1 or keywords2)`

  Source: blusapphire.io
- Example of a condition:

```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 standarizing and converting query to various SIEM platform.

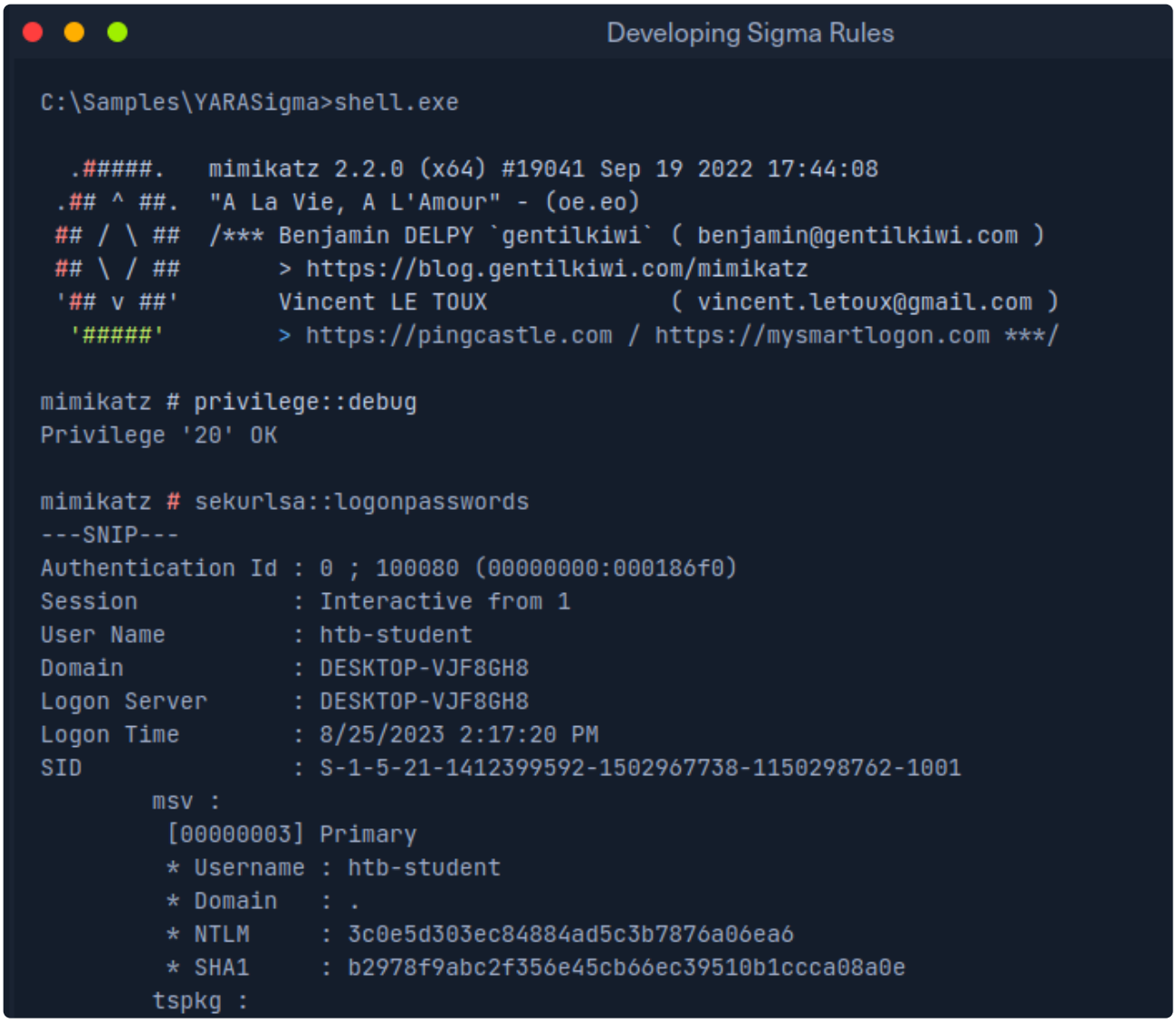
Developing Sigma Rules

Manually Developing a Sigma Rule

Example 1: LSASS Credential Dumping

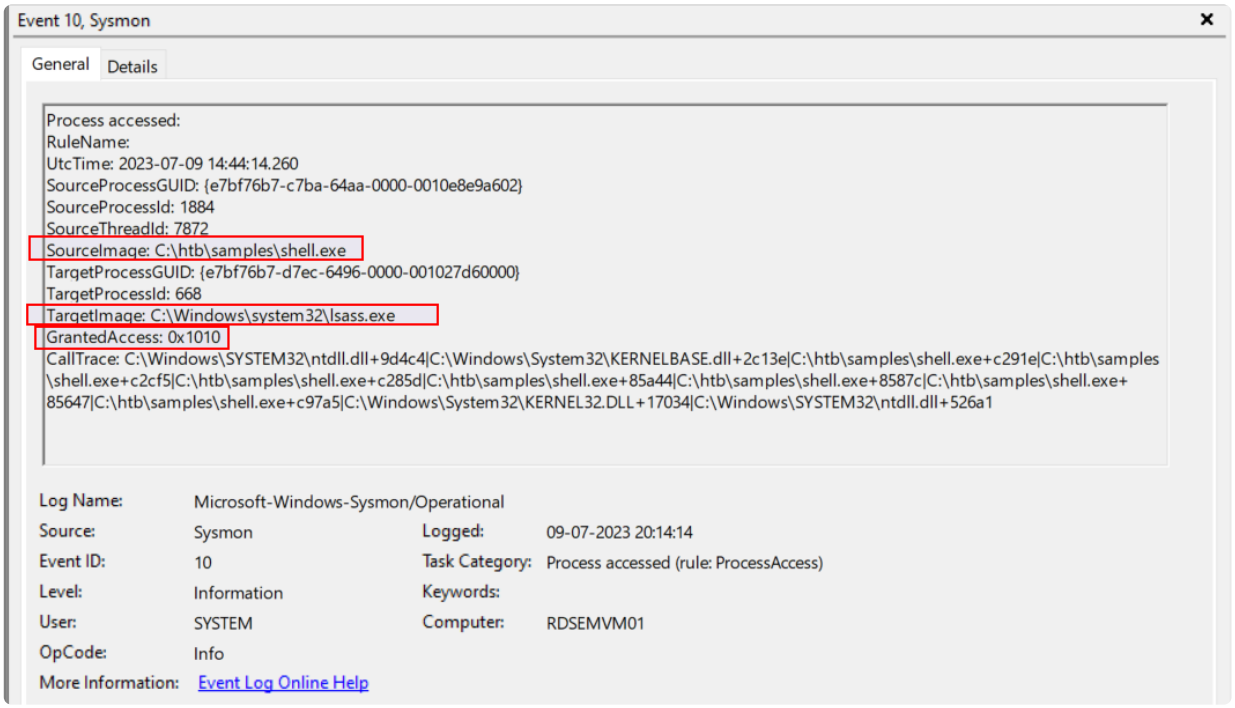
- Let's dive into the world of Sigma rules using a sample named `shell.exe` (a renamed version of mimikatz) residing in the `C:\Samples\YARASigma` directory of this section's target as an illustration.
  - We want to understand the process behind crafting a Sigma rule, so let's get our hands dirty.
- 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
```



- 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.



- 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.

```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.

```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.

```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.

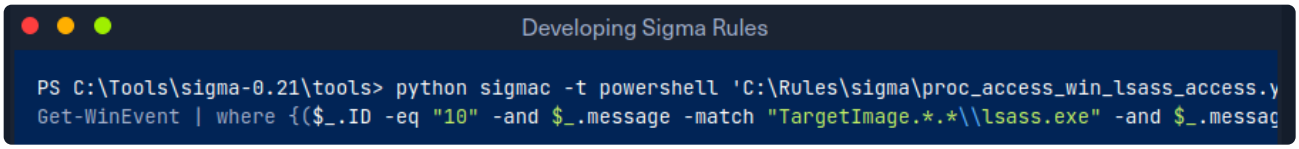
```yaml
logsource:
    category: process_access
    product: windows
```

- `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.

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

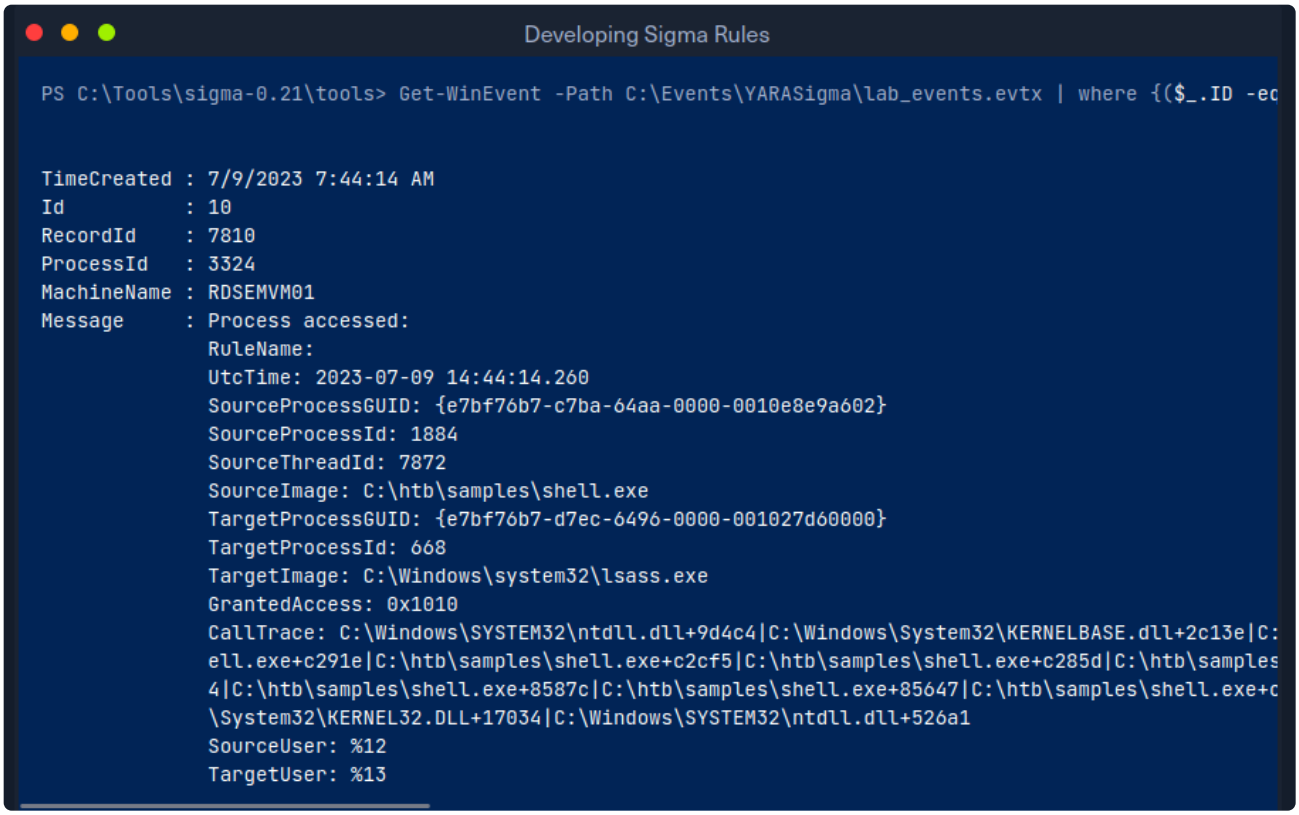
- Our first Sigma rule above can be found inside the `C:\Rules\sigma` directory of this section's target as `proc_access_win_lsass_access.yml`.
  - Let's explore the `sigmac` tool that can help us transform this rule into queries or configurations compatible with a multitude of SIEMs, log management 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
```



- 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
```



- 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
```

```yaml
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/76304f93b390af3bb66e4f451ca16
562a479bdc9/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'
```

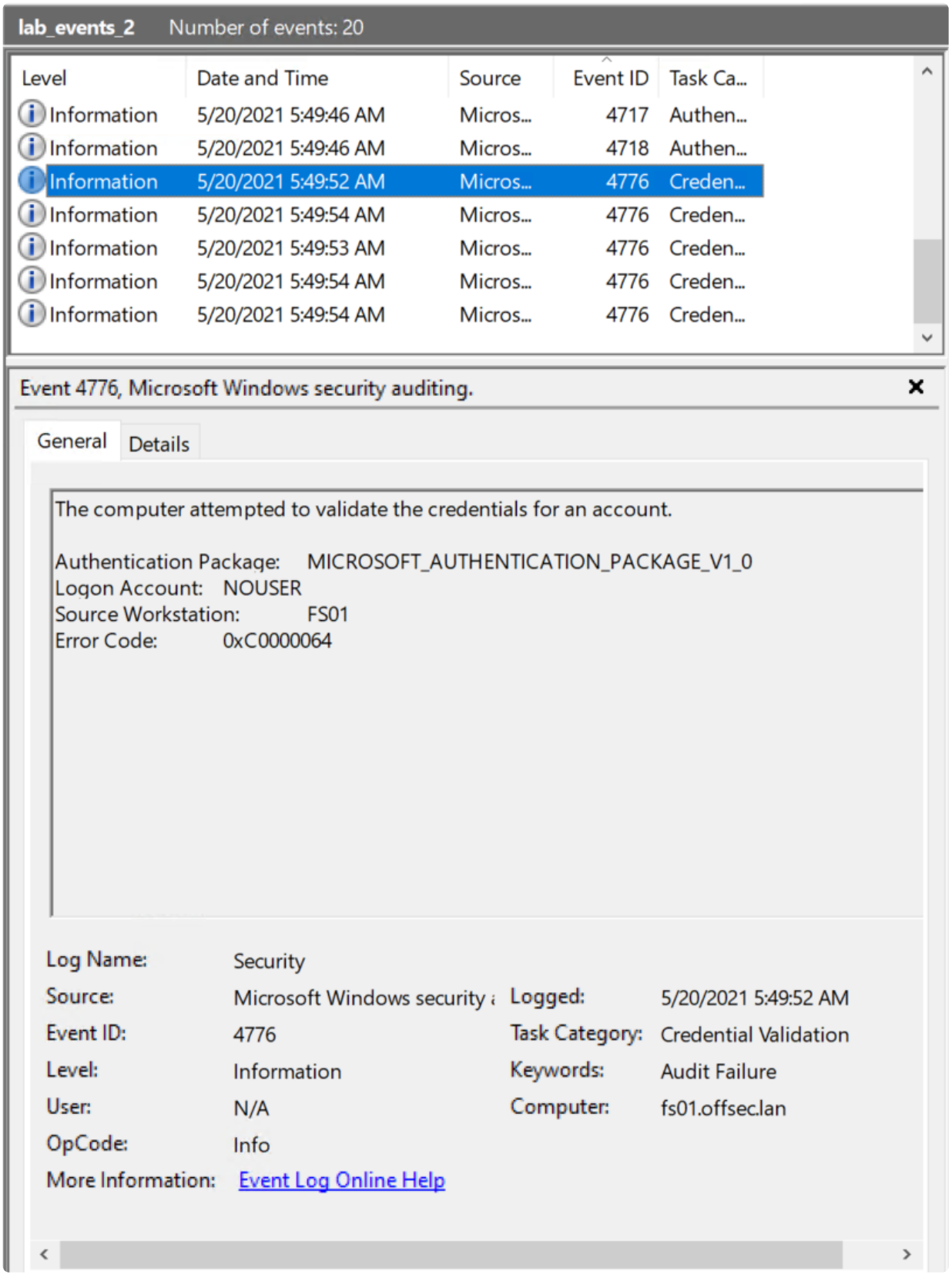
```
        GrantedAccess: '0x1410'
    filter_optional_nextron:
        SourceImage|startswith:
            - 'C:\Windows\Temp\asgard2-agent\'
            - 'C:\Windows\Temp\asgard2-agent-sc\'
        SourceImage|endswith:
            - '\thor64.exe'
            - '\thor.exe'
            - '\aurora-agent-64.exe'
            - '\aurora-agent.exe'
        GrantedAccess:
            - '0x1fffff'
            - '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: '0x1fffff'
    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: '\updater.exe'
        TargetImage|endswith: '\winlogon.exe'
        GrantedAccess: '0x1fffff'
    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'
```

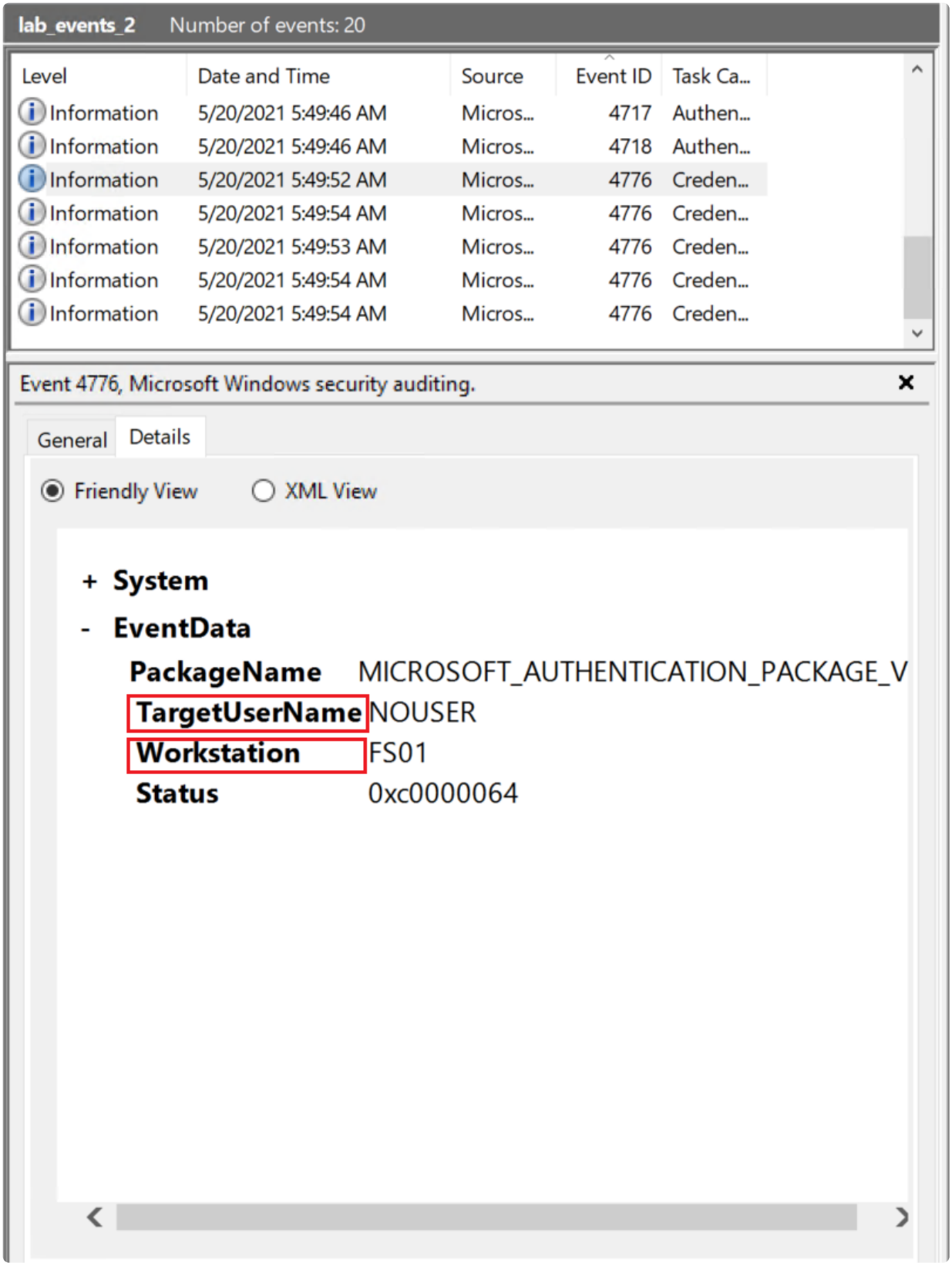
```
        GrantedAccess: '0x1410'
    condition: selection and not 1 of filter_optional_*
fields:
    - User
    - SourceImage
    - GrantedAccess
falsepositives:
    - Updaters and installers are typical false positives. Apply custom
filters depending on your environment
level: medium
```

- 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
Logged: 5/20/2021 5:49:52 AM
Event ID: 4776
Task Category: Credential Validation
Level: Information
Keywords: Audit Failure
User: N/A
Computer: fs01.offsec.lan
OpCode: Info
More Information: Event Log Online Help



- 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`

```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`).

```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.

Sigma Rule Development Resources

- 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 developement.
- 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

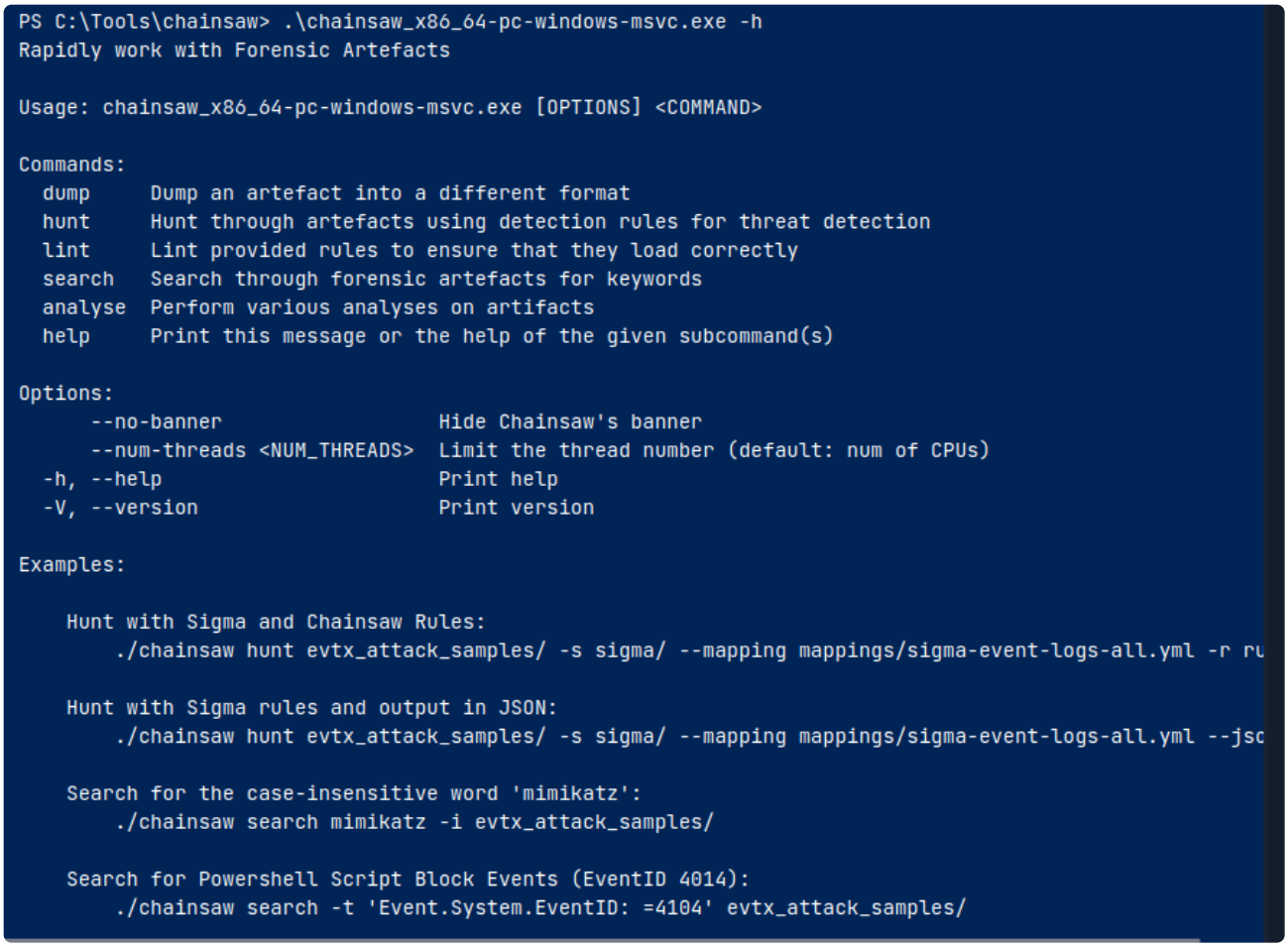
- In cybersecurity, time is of the essence.
  - Rapid analysis allows us to not just identify but also respond to threats before they escalate.
- When we're up against the clock, racing to find a needle in a haystack of Windows Event Logs without access to a SIEM, Sigma rules combined with tools like 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
```



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
```

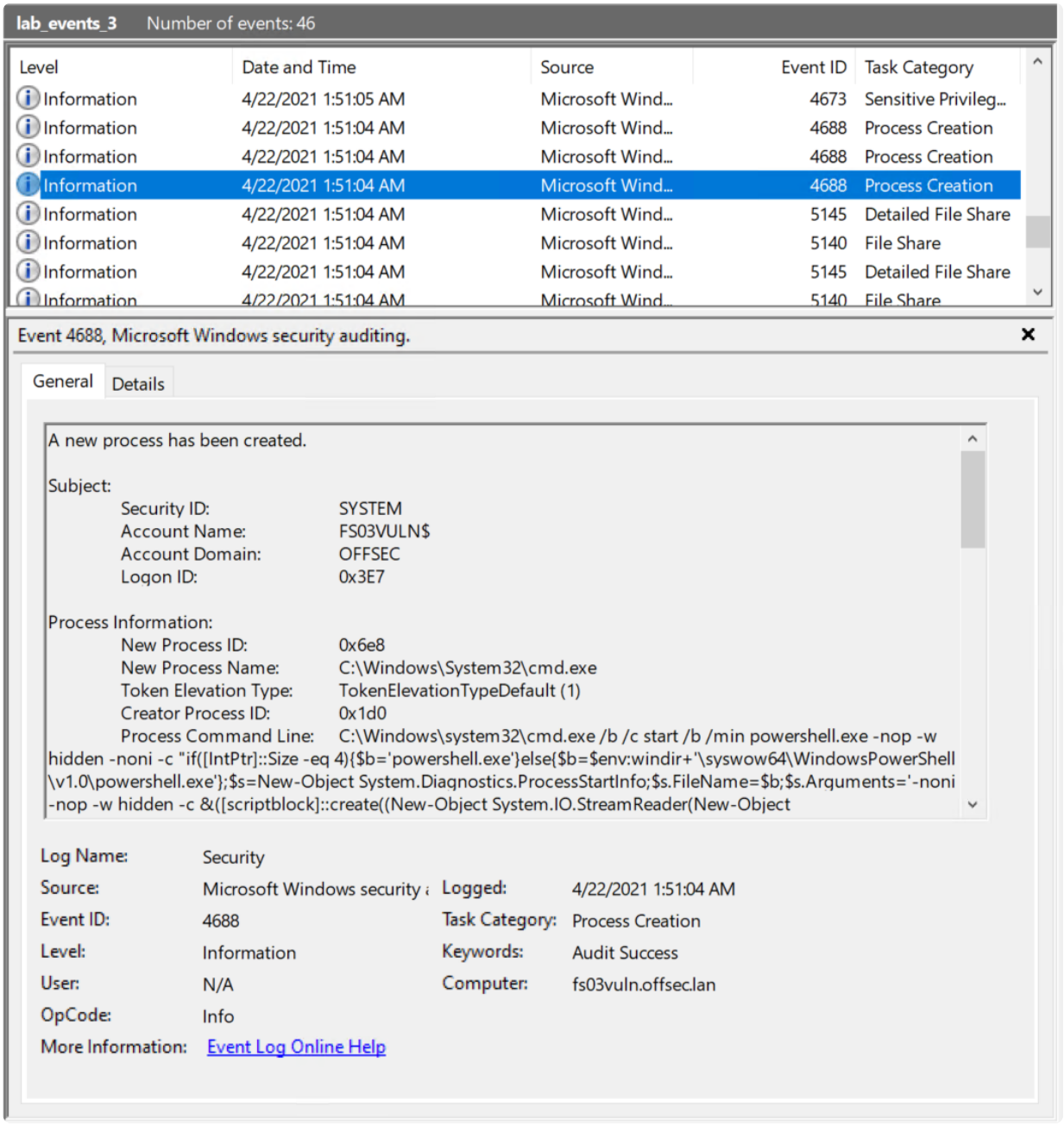
- `detection`: The `selection` section checks if any Windows events with ID 4688 exist and also checks if the `NewProcessName` field ends with `\powershell.exe`, `\pwsh.exe`, or `\cmd.exe`.
  - The `selection_powershell` section checks if the executed command line includes PowerShell-related executables 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
```

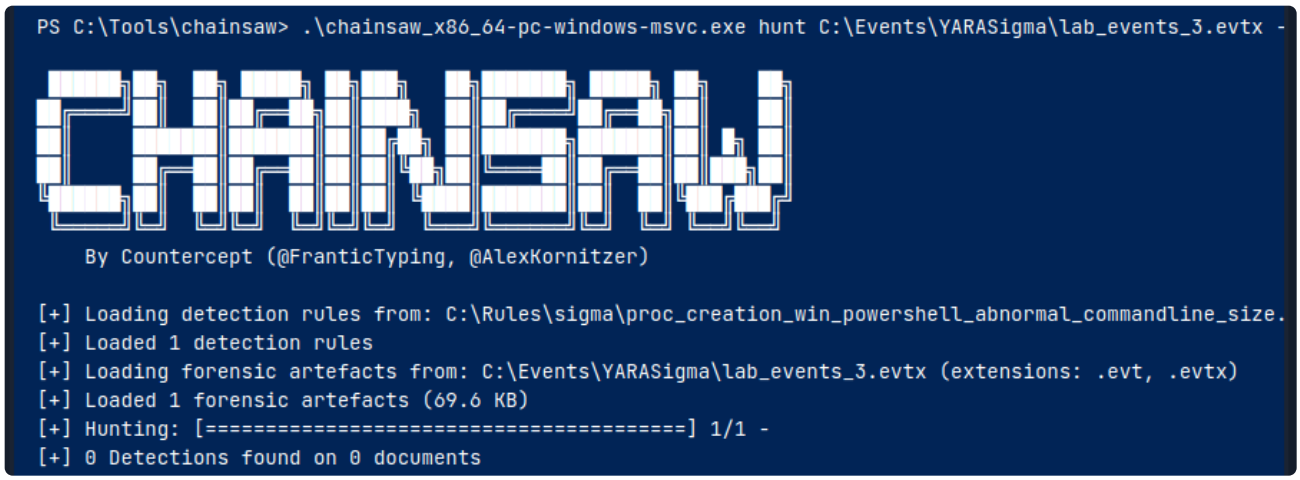
```
        NewProcessName|endswith:
            - '\powershell.exe'
            - '\pwsh.exe'
            - '\cmd.exe'
    selection_powershell:
        CommandLine|contains:
            - 'powershell.exe'
            - 'pwsh.exe'
    selection_length:
        CommandLine|re: '.{1000,}'
    condition: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

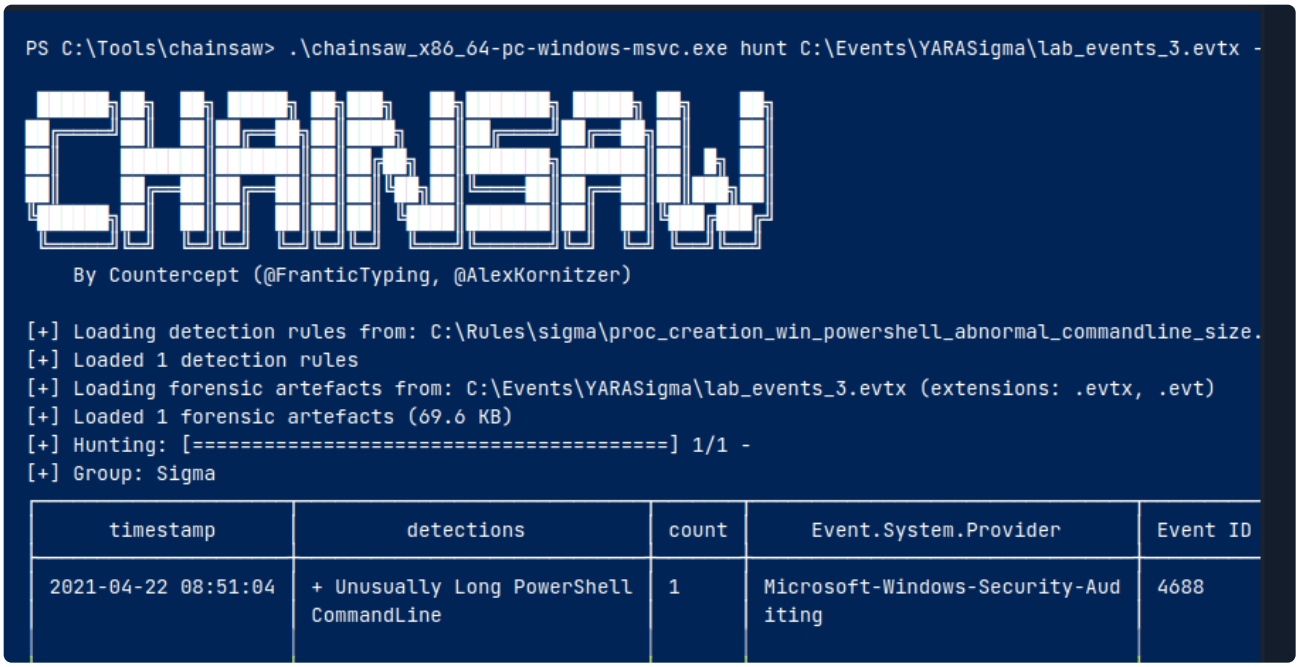



```
PS C:\Tools\chainsaw> .\chainsaw_x86_64-pc-windows-msvc.exe hunt C:\Events\YARASigma\lab_events_3.evtx -s C:\Rules\sigma\proc_creation_win_powershell_abnormal_commandline_size.yml --mapping .\mappings\sigma-event-logs-all.yml
```



- 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.ym
l --mapping .\mappings\sigma-event-logs-all-new.yml
```



- 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 tranlating 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)

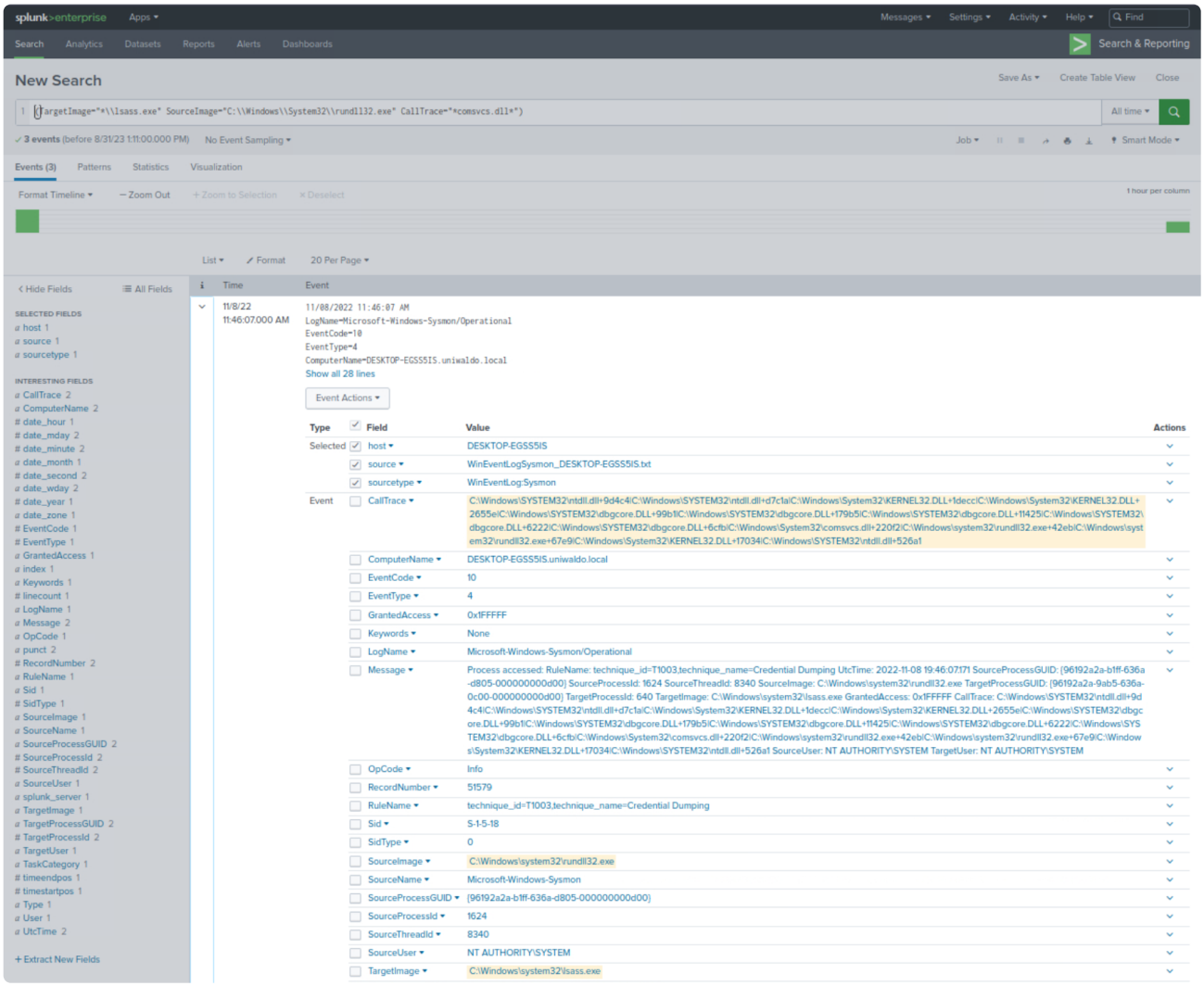
Summary

- 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_lsa
ss_dump_comsvcs_dll.yml -c .\config\splunk-windows.yml

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

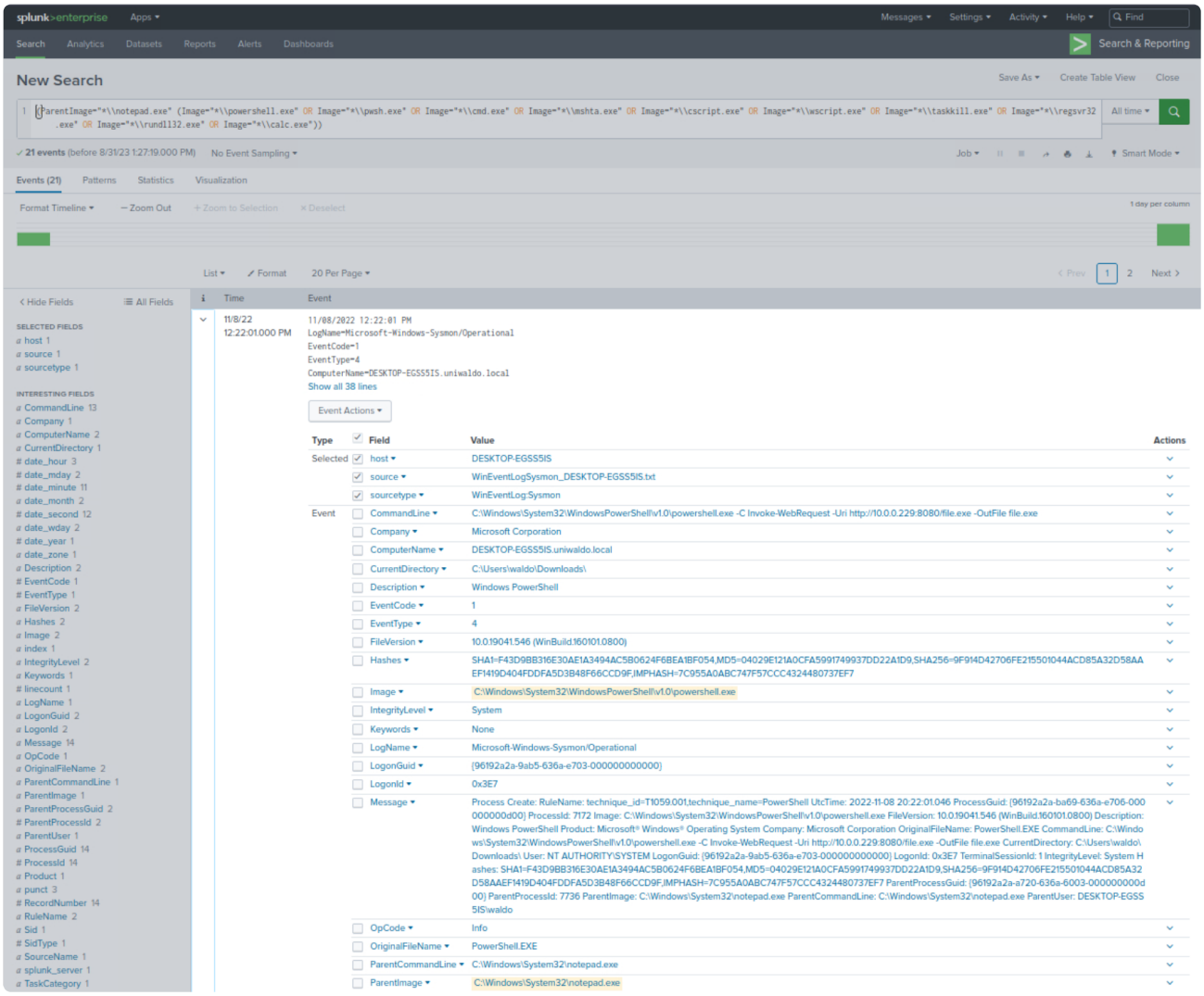


- The Splunk search provided by `sigmac` was indeed able to detect MiniDump function abuse to dump LSASS's memory.

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 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!