yara

Bulid a yara rule example

```
rule my_rule {

    meta:
        author = "Author Name"
        description = "example rule"
        hash = ""

    strings:
        $string1 = "test"
        $string2 = "rule"
        $string3 = "htb"

    condition:
        all of them
}
```

Developing a YARA Rule Through yarGen

```
python3 yarGen.py -m /home/htb-student/temp -o htb_sample.yar
```

Manually Developing a YARA Rule

Neuron Used by Turla

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. (Note: im gonna use dnspy which is better)

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

```
cat code
.assembly extern System.Configuration.Install
{
```

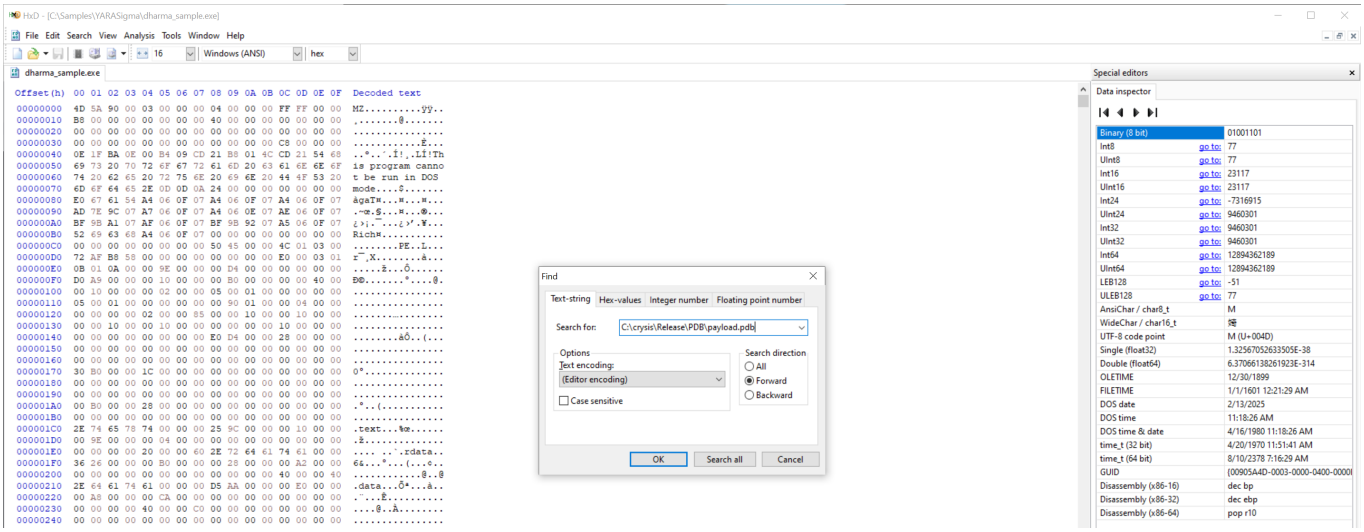
```
  .ver 4:0:0:0
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A ) // .?_....:
}
---SNIP---
```

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

```
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
}
```

Hunting Evil with YARA (Windows Edition)

Hunt for Hex values inside the binary using `HxD` tool



Then write the rule

```
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/bff6a1000a86f8edf3673d576786ec75b80bed0
c458a8ca0bd52d12b74099071/behavior"

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

        condition: all of them
}
```

Hunting for Evil Within Running Processes with YARA (Windows)

```
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
```

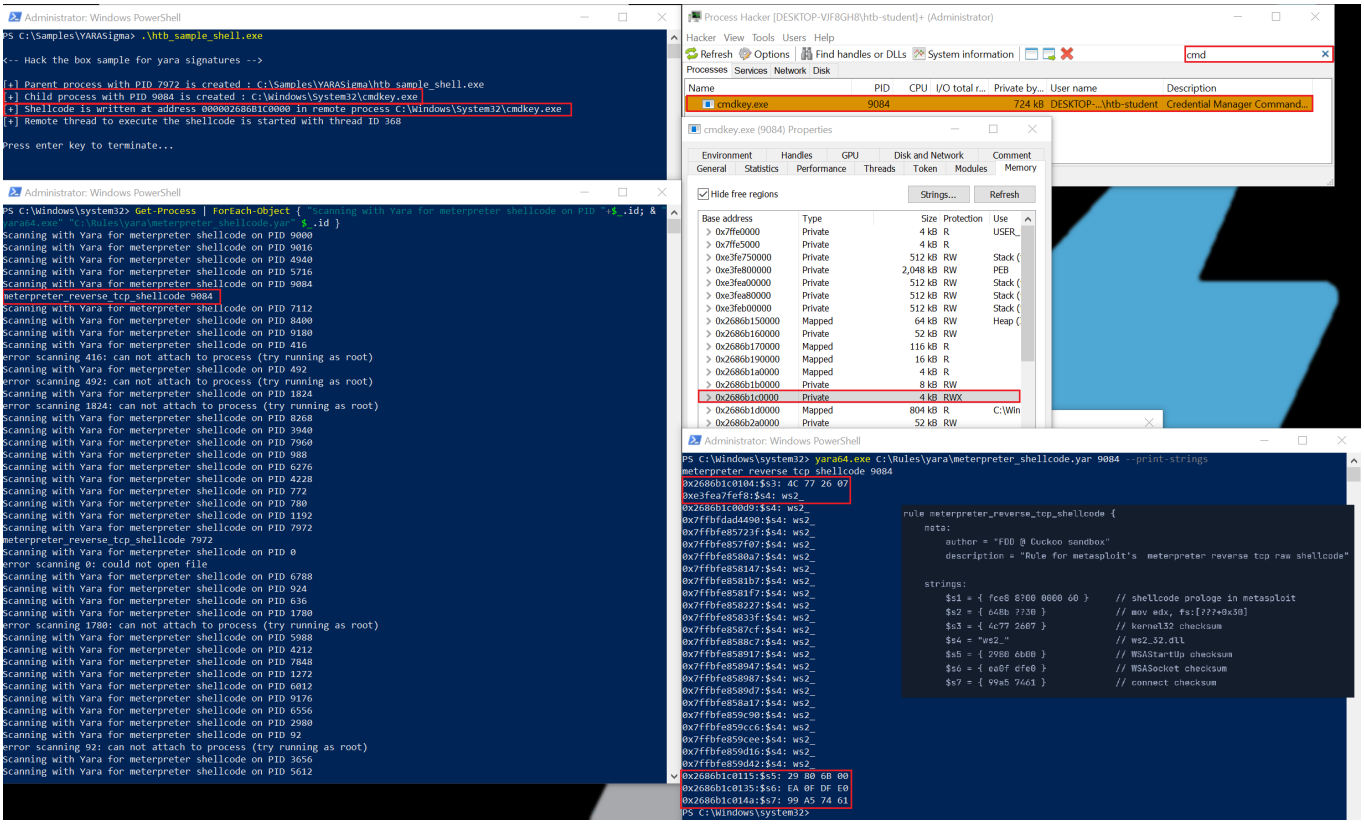
```
        $s2 = { 648b ??30 }             // mov edx, fs:[???+0x30]
        $s3 = { 4c77 2607 }             // kernel32 checksum
        $s4 = "ws2_"                    // ws2_32.dll
        $s5 = { 2980 6b00 }             // WSAStartUp checksum
        $s6 = { ea0f dfe0 }             // WSASocket checksum
        $s7 = { 99a5 7461 }             // connect checksum

    condition:
        5 of them
}
```

Scan for the Process

```
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 }
```

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.

```
yara64.exe C:\Rules\yara\meterpreter_shellcode.yar 9084 --print-strings
```



Hunting for Evil Within ETW Data with YARA

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

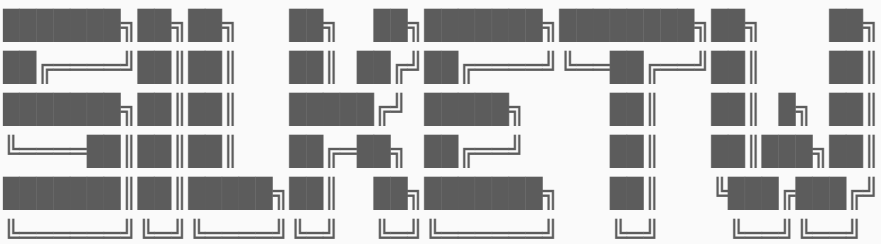
```
PS C:\Tools\SilkETW\v8\SilkETW> .\SilkETW.exe -t user -pn Microsoft-Windows-
PowerShell -ot file -p ./etw_ps_logs.json -l verbose -y C:\Rules\yara  -yo
Matches
```

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*)
}
```

The example based on the role, the tool (SilkETW) will be running with the yara rules and i will execute any of the commands in powershell to see if it get detected

```
Invoke-Command -ScriptBlock {Write-Host "Hello from PowerShell"}
```

```
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



                  [v0.8 - Ruben Boonen => @FuzzySec]

[+] Collector parameter validation success..
[>] Starting trace collector (Ctrl-c to stop)..
[?] Events captured: 28
```

```
      -> Yara match: powershell_hello_world_yara
      -> Yara match: powershell_hello_world_yara
```

Hunting Evil with YARA (Linux Edition)

Hunting for Evil Within Memory Images with YARA

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

OR use vol with the yara plugin

```
vol.py -f /home/htb-student/MemoryDumps/compromised_system.raw yarascan -U
"www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com"
Volatility Foundation Volatility Framework 2.6.1
```

Multiple YARA Rule Scanning Against a Memory Image

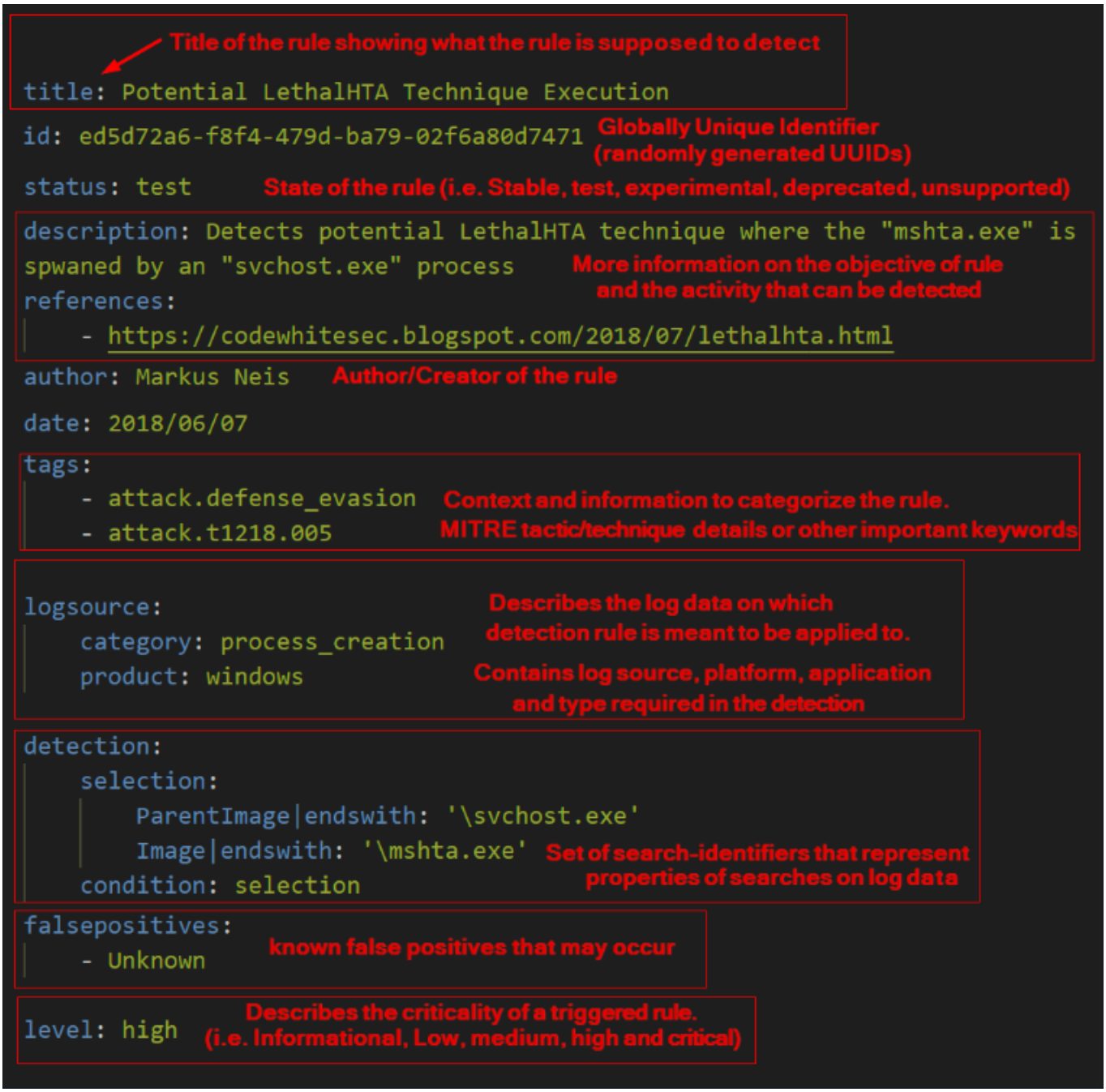
```
ltjax@htb[/htb]$ vol.py -f /home/htb-
student/MemoryDumps/compromised_system.raw yarascan -y /home/htb-
student/Rules/yara/wannacry_artifacts_memory.yar
```

Sigma

Basic Sigma Rule

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

```yaml
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
```



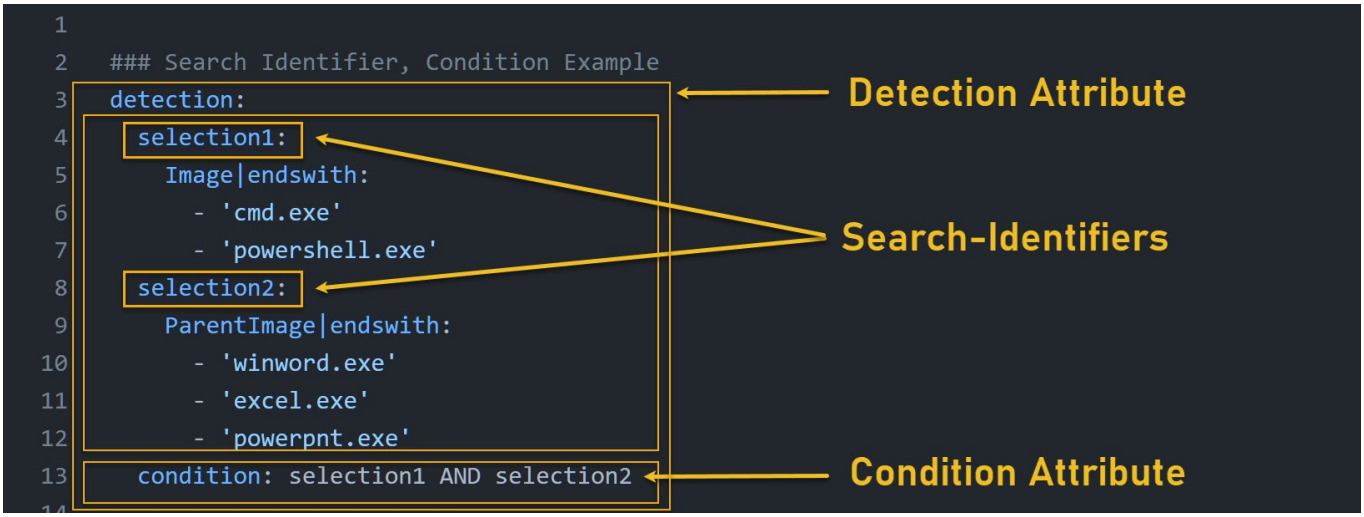
`logsource`: *This section describes the log data on which the detection is meant to be applied to. It describes the log source, the platform, the application and the type that is required in the detection. More information can be found in the following link:* *https://github.com/SigmaHQ/sigma/tree/master/documentation/logsource-guides*

- `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.

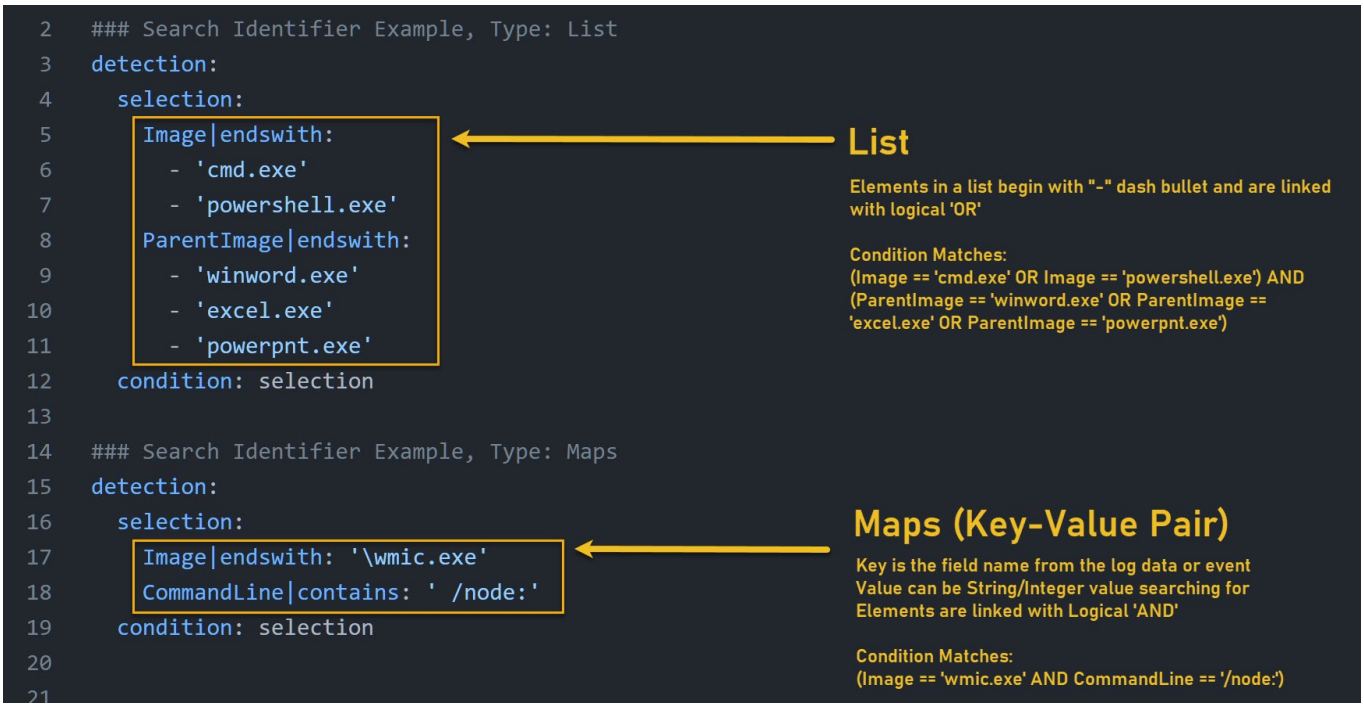
`detection`: A set of search-identifiers that represent properties of searches on log data. Detection is made up of two components:

- Search Identifiers
- Condition



`ists`, 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 `OR`.



Manually Developing a Sigma Rule

Example 1: LSASS Credential Dumping

```
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 flag 0x1010
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
```

Convert the rule to start hunting

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
```

All the above no needed, i could use chainsaw to do this

Hunting Evil with Sigma (Chainsaw Edition)

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

```
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
```