Developing YARA Rules

In this section, we'll cover manual and automated YARA rule development.

Let's now navigate to the bottom of this section and click on "Click here to spawn the target system!". Then, let's SSH into the Target IP using the provided credentials. The vast majority of the actions/commands covered from this point up to end of this section can be replicated inside the target, offering a more comprehensive grasp of the topics presented.

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.

Developing YARA Rules

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

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

Code: yara

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

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

? Go to Questions

Table of Contents

Introduction to YARA & Sigma

Leveraging YARA

YARA and YARA Rules

Developing YARA Rules

Hunting Evil with YARA (Windows Edition)

Hunting Evil with YARA (Linux Edition)

Hunting Evil with YARA (Web Edition)

Leveraging Sigma

Sigma and Sigma Rules

Developing Sigma Rules

Hunting Evil with Sigma (Chainsaw Edition)

Hunting Evil with Sigma (Splunk Edition)

Skills Assessment

Skills Assessment

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left

```
    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. If the rule finds all three strings, it raises an alert, hinting that the file might be packed with the UPX packer. This rule is a handy tool when we're on the lookout for executables that have undergone compression or obfuscation using the UPX method.

Developing a YARA Rule Through yarGen

Let's continue our dive into the world of YARA rules using a sample named `dharma_sample.exe` residing in the `/home/htb-student/Samples/YARASigma` directory of this section's target.

Once again, we need to conduct a string analysis on our malware sample.

Developing YARA Rules

```
MisaelMacias@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.

Here's how we get yarGen up and running:

- Download the latest release from the `release` section
- Install all dependencies with `pip install -r requirements.txt`
- Run `python yarGen.py --update` to automatically download the built-in databases. They will be saved into the `./dbs` subfolder (Download: `913 MB`).
- See help with `python yarGen.py --help` for more information on the command line parameters.

Note: yarGen can be found inside the `/home/htb-student/yarGen-0.23.4` directory of this section's target.

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.

Developing YARA Rules

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

--------------------------------------------------------------------------------
                   _____
    __ _____ _____/ ___/__ ___
   / // / _ `/ __/ (_ / -_) _ \
   \_, /\_,_/_/  \___/\__/_//_/
  /___/  Yara Rule Generator
         Florian Roth, July 2020, Version 0.23.3

  Note: Rules have to be post-processed
  See this post for details: https://medium.com/@cyb3rops/121d29322282
--------------------------------------------------------------------------------
[+] Using identifier 'temp'
[+] Using reference 'https://github.com/Neo23x0/yarGen'
[+] Using prefix 'temp'
[+] Processing PEStudio strings ...
[+] Reading goodware strings from database 'good-strings.db' ...
    (This could take some time and uses several Gigabytes of RAM depending on your db size)
[+] Loading ./dbs/good-imphashes-part3.db ...
[+] Total: 4029 / Added 4029 entries
[+] Loading ./dbs/good-strings-part9.db ...
[+] Total: 788 / Added 788 entries
[+] Loading ./dbs/good-strings-part8.db ...
[+] Total: 332082 / Added 331294 entries
[+] Loading ./dbs/good-imphashes-part4.db ...
[+] Total: 6426 / Added 2397 entries
[+] Loading ./dbs/good-strings-part2.db ...
[+] Total: 1703601 / Added 1371519 entries
[+] Loading ./dbs/good-exports-part2.db ...
[+] Total: 90960 / Added 90960 entries
[+] Loading ./dbs/good-strings-part4.db ...
[+] Total: 3860655 / Added 2157054 entries
[+] Loading ./dbs/good-exports-part4.db ...
[+] Total: 172718 / Added 81758 entries
[+] Loading ./dbs/good-exports-part7.db ...
[+] Total: 223584 / Added 50866 entries
[+] Loading ./dbs/good-strings-part6.db ...
[+] Total: 4571266 / Added 710611 entries
[+] Loading ./dbs/good-strings-part7.db ...
[+] Total: 5828908 / Added 1257642 entries
[+] Loading ./dbs/good-exports-part1.db ...
[+] Total: 293752 / Added 70168 entries
[+] Loading ./dbs/good-exports-part3.db ...
[+] Total: 326867 / Added 33115 entries
[+] Loading ./dbs/good-imphashes-part9.db ...
[+] Total: 6426 / Added 0 entries
[+] Loading ./dbs/good-exports-part9.db ...
[+] Total: 326867 / Added 0 entries
[+] Loading ./dbs/good-imphashes-part5.db ...
[+] Total: 13764 / Added 7338 entries
[+] Loading ./dbs/good-imphashes-part8.db ...
[+] Total: 13947 / Added 183 entries
[+] Loading ./dbs/good-imphashes-part6.db ...
```

```
[+] Total: 13976 / Added 29 entries
[+] Loading ./dbs/good-strings-part1.db ...
[+] Total: 6893854 / Added 1064946 entries
[+] Loading ./dbs/good-imphashes-part7.db ...
[+] Total: 17382 / Added 3406 entries
[+] Loading ./dbs/good-exports-part6.db ...
[+] Total: 328525 / Added 1658 entries
[+] Loading ./dbs/good-imphashes-part2.db ...
[+] Total: 18208 / Added 826 entries
[+] Loading ./dbs/good-exports-part8.db ...
[+] Total: 332359 / Added 3834 entries
[+] Loading ./dbs/good-strings-part3.db ...
[+] Total: 9152616 / Added 2258762 entries
[+] Loading ./dbs/good-strings-part5.db ...
[+] Total: 12284943 / Added 3132327 entries
[+] Loading ./dbs/good-imphashes-part1.db ...
[+] Total: 19764 / Added 1556 entries
[+] Loading ./dbs/good-exports-part5.db ...
[+] Total: 404321 / Added 71962 entries
[+] Processing malware files ...
[+] Processing /home/htb-student/temp/dharma_sample.exe ...
[+] Generating statistical data ...
[+] Generating Super Rules ... (a lot of magic)
[+] Generating Simple Rules ...
[-] Applying intelligent filters to string findings ...
[-] Filtering string set for /home/htb-student/temp/dharma_sample.exe ...
[=] Generated 1 SIMPLE rules.
[=] All rules written to htb_sample.yar
[+] yarGen run finished
```

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.

Developing YARA Rules

```
MisaelMacias@htb[/htb]$ cat htb_sample.yar
/*
   YARA Rule Set
   Author: yarGen Rule Generator
   Date: 2023-08-24
   Identifier: temp
   Reference: https://github.com/Neo23x0/yarGen
*/

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

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 = "Wylz@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.

Developing YARA Rules

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

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

Manually Developing a YARA Rule

Example 1: ZoxPNG RAT Used by APT17

Let's now go a bit deeper...

We want to develop a YARA rule to scan for a specific variation of the `ZoxPNG` RAT used by `APT17` based on:

- A sample named `legit.exe` residing in the `/home/htb-student/Samples/YARASigma` directory of this section's target
- A post from Intezer
- String analysis
- Imphash
- Common sample file size

Let's start with our string analysis endeavors as follows.

Developing YARA Rules

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

 inflate 1.1.4 Copyright 1995-2002 Mark Adler
Sleep
LocalAlloc
CloseHandle
GetLastError
VirtualFree
VirtualAlloc
GetProcAddress
LoadLibraryA
GetCurrentProcessId
GlobalMemoryStatusEx
GetCurrentProcess
GetACP
GetVersionExA
GetComputerNameA
GetTickCount
GetSystemTime
```

```
LocalFree
CreateProcessA
CreatePipe
TerminateProcess
ReadFile
PeekNamedPipe
WriteFile
SetFilePointer
CreateFileA
GetFileSize
GetDiskFreeSpaceExA
GetDriveTypeA
GetLogicalDriveStringsA
CreateDirectoryA
FindClose
FindNextFileA
FindFirstFileA
MoveFileExA
OpenProcess
KERNEL32.dll
LookupAccountSidA
ADVAPI32.dll
SHFileOperationA
SHELL32.dll
strcpy
rand
sprintf
memcpy
strncpy
srand
_snprintf
atoi
strcat
strlen
printf
memset
strchr
memcmp
MSVCRT.dll
_exit
_XcptFilter
exit
__p___initenv
__getmainargs
_initterm
__setusermatherr
_adjust_fdiv
__p__commode
__p__fmode
__set_app_type
_except_handler3
_controlfp
InternetCrackUrlA
InternetCloseHandle
InternetReadFile
HttpQueryInfoA
HttpSendRequestA
InternetSetOptionA
HttpAddRequestHeadersA
HttpOpenRequestA
InternetConnectA
InternetOpenA
WININET.dll
ObtainUserAgentString
urlmon.dll
WTSFreeMemory
WTSEnumerateProcessesA
WTSAPI32.dll
GetModuleFileNameExA
PSAPI.DLL
calloc
free
http://%s/imgres?q=A380&hl=en-US&sa=X&biw=1440&bih=809&tbm=isus&tbnid=aLW4-J8Q1lmYBM:&imgrefurl=htt
http://0.0.0.0/1
http://0.0.0.0/2
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2; .NETCLR 2.0.50727)
image/pjpeg
image/jpeg
image/x-xbitmap
image/gif
Content-Type: application/x-www-form-urlencoded
B64:[%s]
Step 11
Step 10
```

```
Step 9
Step 8
Step 7
Step 6
Content-Type: image/x-png
Step 5
Step 4
Connection: close
Accept-Encoding: gzip, deflate
Accept-Language: en-US
Pragma: no-cache
User-Agent:
Cookie: SESSIONID=%s
Step 3
HTTP/1.1
Step 2
POST
Step 1
Get URL Info Error
[IISEND=0x%08X][Recv:] 0x%08X %s
IISCMD Error:%d
hWritePipe2 Error:%d
kernel32.dll
QueryFullProcessImageName
Not Support This Function!
1.1.4
need dictionary
incorrect data check
incorrect header check
invalid window size
unknown compression method
incompatible version
buffer error
insufficient memory
data error
stream error
file error
stream end
invalid bit length repeat
too many length or distance symbols
invalid stored block lengths
invalid block type
invalid distance code
invalid literal/length code
incomplete dynamic bit lengths tree
oversubscribed dynamic bit lengths tree
incomplete literal/length tree
oversubscribed literal/length tree
empty distance tree with lengths
incomplete distance tree
oversubscribed distance tree
Z0X03
>0!0
Western Cape1
Durbanville1
Thawte1
Thawte Certification1
Thawte Timestamping CA0
121221000000Z
201230235959Z0^1
Symantec Corporation100.
'Symantec Time Stamping Services CA - G20
"W*o
]jxdE
`F~T
&0$0"
http://ocsp.thawte.com0
80604
.http://crl.thawte.com/ThawteTimestampingCA.crl0
TimeStamp-2048-10
y@b%
thawte, Inc.1(0&
Certification Services Division1806
/(c) 2006 thawte, Inc. - For authorized use only1
thawte Primary Root CA0
061117000000Z
36071623595 9Z0
thawte, Inc.1(0&
Certification Services Division1806
/(c) 2006 thawte, Inc. - For authorized use only1
thawte Primary Root CA0
l[HhIY7
tf/j8
S}+
```

```
>n)i
B0@0
WHP0
Thawte, Inc.1$0"
Thawte Code Signing CA - G20
110622000000Z
1307212359592Z0
Seoul1
Seongdong-gu1
        4NB Corp.1"0
tigation Development Team1
        4NB Corp.0
LO'%
oWx6
IB-8l3
40200
*http://cs-g2-crl.thawte.com/ThawteCSG2.crl0
&0$0"
http://ocsp.thawte.com0
}1JBAe
^b/1
a{b2
Ti,[\{
thawte, Inc.1(0&
Certification Services Division1806
/(c) 2006 thawte, Inc. - For authorized use only1
thawte Primary Root CA0
1002080000000Z
2002072359592Z0J1
Thawte, Inc.1$0"
Thawte Code Signing CA - G20
,p&7E
rqD=X
n8}v
-0+0)
#http://crl.thawte.com/ThawtePCA.crl0
&0$0"
http://ocsp.thawte.com0
VeriSignMPKI-2-100
---SNIP---
Symantec Corporation100.
'Symantec Time Stamping Services CA - G20
1210180000000Z
2012292359592Z0b1
Symantec Corporation1402
+Symantec Time Stamping Services Signer - G40
2oNW
a;EQ
g0e0*
http://ts-ocsp.ws.symantec.com07
+http://ts-aia.ws.symantec.com/tss-ca-g2.cer0<
50301
+http://ts-crl.ws.symantec.com/tss-ca-g2.crl0(
TimeStamp-2048-20
---SNIP---
Thawte, Inc.1$0"
Thawte Code Signing CA - G2
1(0&
www.4nb.co.kr 0
#Ak_
$>X[hL
0r0^1
Symantec Corporation100.
'Symantec Time Stamping Services CA - G2
130529085845Z0#
*PU19
mOLT
}Jdf6
K%&J
(E~Q
Eev7aSp
```

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.

```
MisaelMacias@htb[/htb]$ python3 imphash_calc.py /home/htb-student/Samples/YARASigma/legit.exe
414bbd566b700ea021cfae3ad8f4d9b9
```

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`.

Code: yara

```
/*
   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/L
      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
      $x2 = "http://%s/imgres?q=A380&hl=en-US&sa=X&biw=1440&bih=809&tbm=isus&tbnid=aLW4-J8Q1lmYBM"

      $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.

Developing YARA Rules

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

Developing YARA Rules

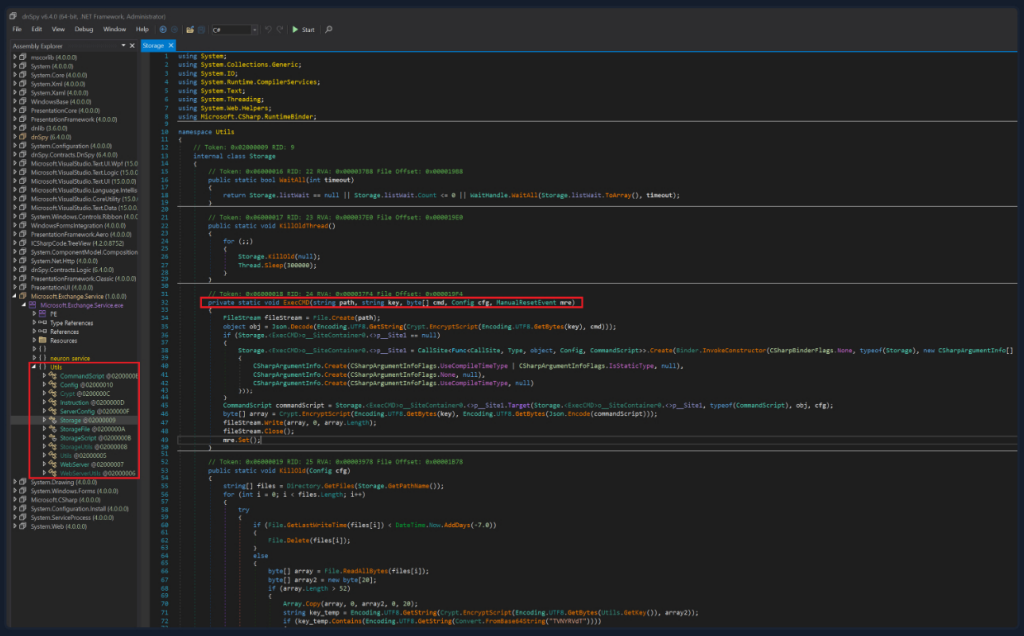
```
MisaelMacias@htb[/htb]$ cat code
.assembly extern System.Configuration.Install
{
  .ver 4:0:0:0
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A ) // .?_....:
}
---SNIP---
  .class public auto ansi abstract sealed beforefieldinit StorageUtils
  } // end of class Utils.StorageUtils
---SNIP---
          default void ExecCMD (string path, string key, unsigned int8[] cmd, class Utils.Config c
        IL_0028:  ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscor
        IL_0070:  stsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscor
        IL_0075:  ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscor
        IL_007f:  ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1<class [mscor
    } // 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 Uti
---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 i
        IL_0eff:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned i
        IL_0f4e:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned i
        IL_0fec:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned i
        IL_102f:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned i
        IL_0018:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned i
        IL_00b1:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned i
          IL_009a:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned
          IL_0142:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned
          default unsigned int8[] EncryptScript (unsigned int8[] pwd, unsigned int8[] data)  cil m
    } // end of method Crypt::EncryptScript
            IL_00a0:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsign
          IL_0052:  call unsigned int8[] class Utils.Crypt::EncryptScript(unsigned int8[], unsigned
---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`.

Code: yara

```
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.

Developing YARA Rules

```
MisaelMacias@htb[/htb]$ python3 entropy_pe_section.py -f /home/htb-student/Samples/YARASigma/sham2.
        virtual address: 0x1000
        virtual size: 0x25f86
        raw size: 0x26000
        entropy: 6.4093453613451885
.rdata
```

```
        virtual address: 0x27000
        virtual size: 0x62d2
        raw size: 0x6400
        entropy: 4.913675128870228
.data
        virtual address: 0x2e000
        virtual size: 0xb744
        raw size: 0x9000
        entropy: 1.039771174750106
.rsrc
        virtual address: 0x3a000
        virtual size: 0xc888
        raw size: 0xca00
        entropy: 7.976847940518103
```

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

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

Code: yara

```
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
  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_resour
( (math.entropy(pe.resources[i].offset, pe.resources[i].length) > 7.8) and pe.resources[i].id == 10
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

- 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:

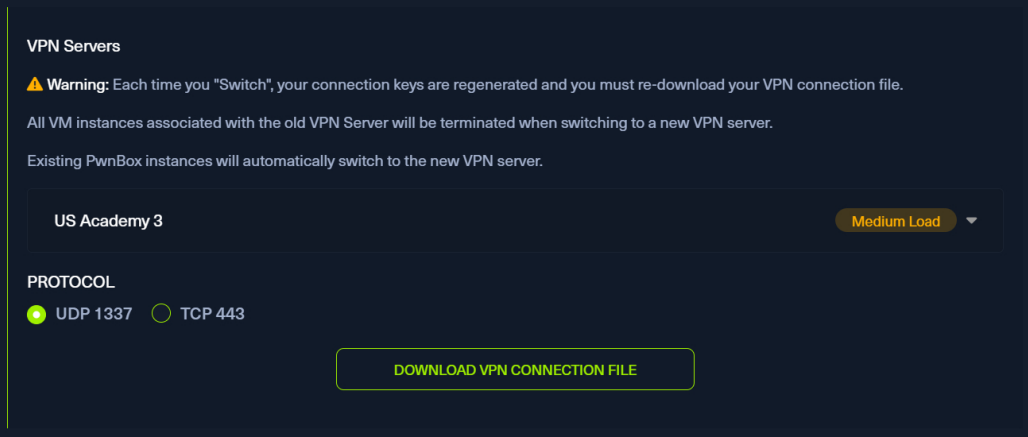
- How to Write Simple but Sound Yara Rules - Part 1
- How to Write Simple but Sound Yara Rules - Part 2
- How to Write Simple but Sound Yara Rules - Part 3

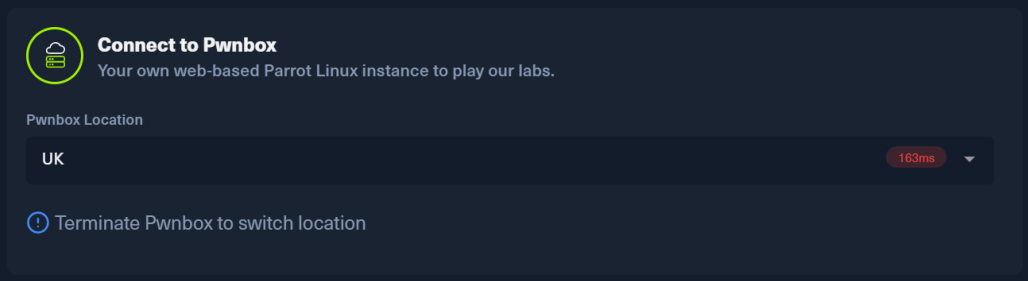
`yarGen` is a great tool to develop some good yara rules by extracting unique patterns. Once the rule is developed with the help of yarGen, we definitely need to review and add/remove some more patterns to make it an effective rule.

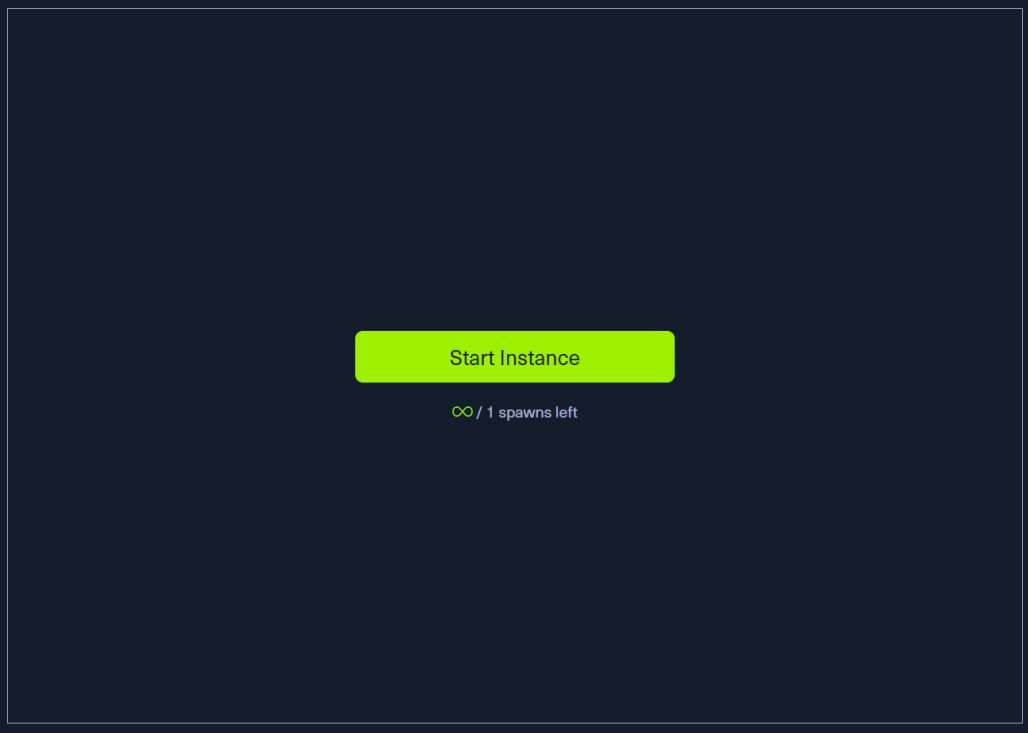
In this blogpost, Florian Roth has mentioned that the main purpose of yarGen is to develop the best possible rules for

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.

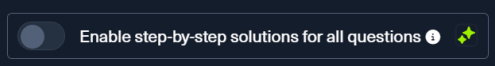
Continuing forward, we'll dig deeper into using YARA, expanding our hunt for threats from our filesystem to memory and also within memory images.







Waiting to start...



Questions

Answer the question(s) below to complete this Section and earn cubes!

Download VPN Connection File

Target(s): Click here to spawn the target system!

SSH to with user "htb-student" and password "HTB_@cademy_stdnt!"

+ 2 Perform string analysis on the "DirectX.dll" sample that resides in the "/home/htb-student/Samples/YARASigma" directory of this section's target. Then, study the "apt_apt17_mal_sep17_1.yar" YARA rule that resides in the "/home/htb-student/Rules/yara" directory and replace "X.dll" with the correct DLL name to ensure the rule will identify "DirectX.dll". Enter the correct DLL name as your answer. Answer format: _.dll

TSMSISrv.dll

Submit

Previous Next

Mark Complete & Next