Mastering YARA Rules: A Step-by-Step Guide

Learn to Hunt Malware with Precision

Sara Khanchi
INCS 745 – NYIT

Outline

- Introduction to YARA
- Anatomy of a YARA Rule
- Writing YARA Rule
  - Using Wildcards and Placeholders
  - Matching Multiple Strings with Conditions
  - Matching Assembly Code and Hex Patterns
  - Regular Expressions in YARA
- Conclusion

What is YARA? Why Use It?

- YARA = Yet Another Recursive Acronym
- "The pattern matching swiss knife for malware researchers (and everyone else)"
- Used in antivirus engines, threat hunting, incident response
- Hosted on GitGub http://plusvic.github.io/yara/
- Pattern matching:
    - strings (ASCII, UCS-2)
    - regular expressions
    - binary patterns (hex strings)
- Classification:
    - on input: combination of strings
    - on output: tags, metadata

Introduction

- What YARA is NOT
  - Not a virus scanner
  - Not a correlation engine
  - Not a bayesian classifier
  - No artifical intelligence (AI) involved

Introduction

- What YARA IS
  - A "better grep"
- Use cases:
  - Finding interesting entries on pastebin.com ...
  - Triage data
  - Preprocess files to direct reverse engineering efforts
- Integrate it into your projects:
  - C library
  - Python bindings https://github.com/plusvic/yara/tree/master/yara-python
  - Ruby bindings https://github.com/SpiderLabs/yara-ruby

Introduction

- YARA rules are supported by security products and services
  - FireEye appliances
  - Fidelis XPS
  - RSA ECAT
  - Volatility
  - ThreadConnect threat intelligence exchange
  - VirusTotal Intelligence
  - ...

Writing YARA Rules

Anatomy of a Simple YARA Rule

```
rule my_example : tag1 tag2 tag3
{
  meta:
        description = "This is just an example"
        thread_level = 3
        in_the_wild = true
  strings:
        $a = { 6A 40 68 00 30 00 00 6A 14 8D 91 }
        $b = /[0-9a-f]{32}/
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
  condition:
        $a or ( $b and $c)
}
```

Yara Options

```
$ yara
usage:  yara [OPTION]... [RULEFILE]... FILE
options:
  -t <tag>                  print rules tagged as <tag> and ignore the
                            rest. Can be used more than once.
  -i <identifier>           print rules named <identifier> and ignore the
                            rest. Can be used more than once.
  -n                        print only not satisfied rules (negate).
  -g                        print tags.
  -m                        print metadata.
  -s                        print matching strings.
  -d <identifier>=<value>   define external variable.
  -r                        recursively search directories.
  -f                        fast matching mode.
  -v                        show version information.
```

Yara Rule-Editor to Use

- Editors you can use
  - vim (with simple syntax highlighting)
  - gvim (with GUI and syntax highlighting)
  - Emacs
  - gedit
  - Visual Studio
  - Notepad++

Hello World!

Your First YARA Rule

Your First Yara Rule

- Create a file named "hello.yara" with the following contents:

```
rule Hello_World
{
        condition: true
}
```

- Now let the computer greet you:

```
$ yara hello.yara /yara/malware/somefile.txt
```

Note: "/yara/malware/somefile.txt" is one file in your system

Another Yara Rule

- Change the "hello.yara" with the following contents:

```
rule HelloWorld
{
    strings:
        $a = "Hello, world!"
    condition:
        $a
}
```

- Now let the computer greet you:

```
$ yara hello.yara /yara/malware/somefile.txt
```

Note: "/yara/malware/somefile.txt" is one file in your system

Yara Rule-Input from User

- Create a file named "greeting.yara" with the following contents:

```
rule GoodMorning
{
        condition:
                hour < 12 and hour >= 4
}
```

- Create another rule for the GoodEvening greetings

- Now pass different values for "hour" to the rule set:

```
$ yara -d hour=8 greeting.yara /yara/malware/somefile.txt
GoodMorning /yara/files/somefile.txt

$ yara -d hour=20 greeting.yara /yara/malware/somefile.txt
GoodEvening /yara/files/somefile.txt
```

Yara Rule-Counting Strings

- Use # to count string matches:

```
rule ManyOccurrences
{
    strings:
        $a = "malicious"
    condition:
        #a > 5
}
```

```
$ yara rules.yara /yara/malware/somefile.txt
```

Detecting File Types

Detecting File Types-Executables

- A simple specification for PE files (Windows executable)
- Task: To find any files in Portable Executable ("PE") format
- Simple specification: File must contain the strings "MZ" and "PE"

```
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 c8 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  65 cd 43 c7 21 ac 2d 94  21 ac 2d 94 21 ac 2d 94  |e.C.!.-.!.-.!.-.|
00000090  21 ac 2c 94 25 ac 2d 94  e2 a3 70 94 24 ac 2d 94  |!.,.%.-...p.$.-.|
000000a0  c9 b3 26 94 23 ac 2d 94  52 69 63 68 21 ac 2d 94  |..&.#.-.Rich!.-.|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  50 45 00 00 4c 01 03 00  |........PE..L...|
```

Identify Executable Files

- Create a new file, named “executable.yara”
- Start with a blank rule:

```
rule PE_file
{
}
```

Identify Executable Files

- Now add the two strings:

```
rule PE_file
{
        strings:
                $mz = "MZ"
                $pe = "PE"

}
```

- Note: Strings are case-sensitive by default!

Identify Executable Files

- A portable executable file MUST contain both strings. So, add the proper condition:

```
rule PE_file
{
        strings:
                $mz = "MZ"
                $pe = "PE"
        condition:
                $mz and $pe
}
```

- Test your rule file:

```
$ yara -r executable.yara /yara/malware
```

Identify Executable Files

- More constraints:
  - "MZ" at offset 0
  - UInt32 at offset 0x3c points to "PE"
- Refine your condition section:

```
condition:
        ($mz at 0) and
        ($pe at (uint32(0x3c)))
```

- Test your rule file again:

```
$ yara -r executable.yara /yara/malware
```

Identify Executable Files

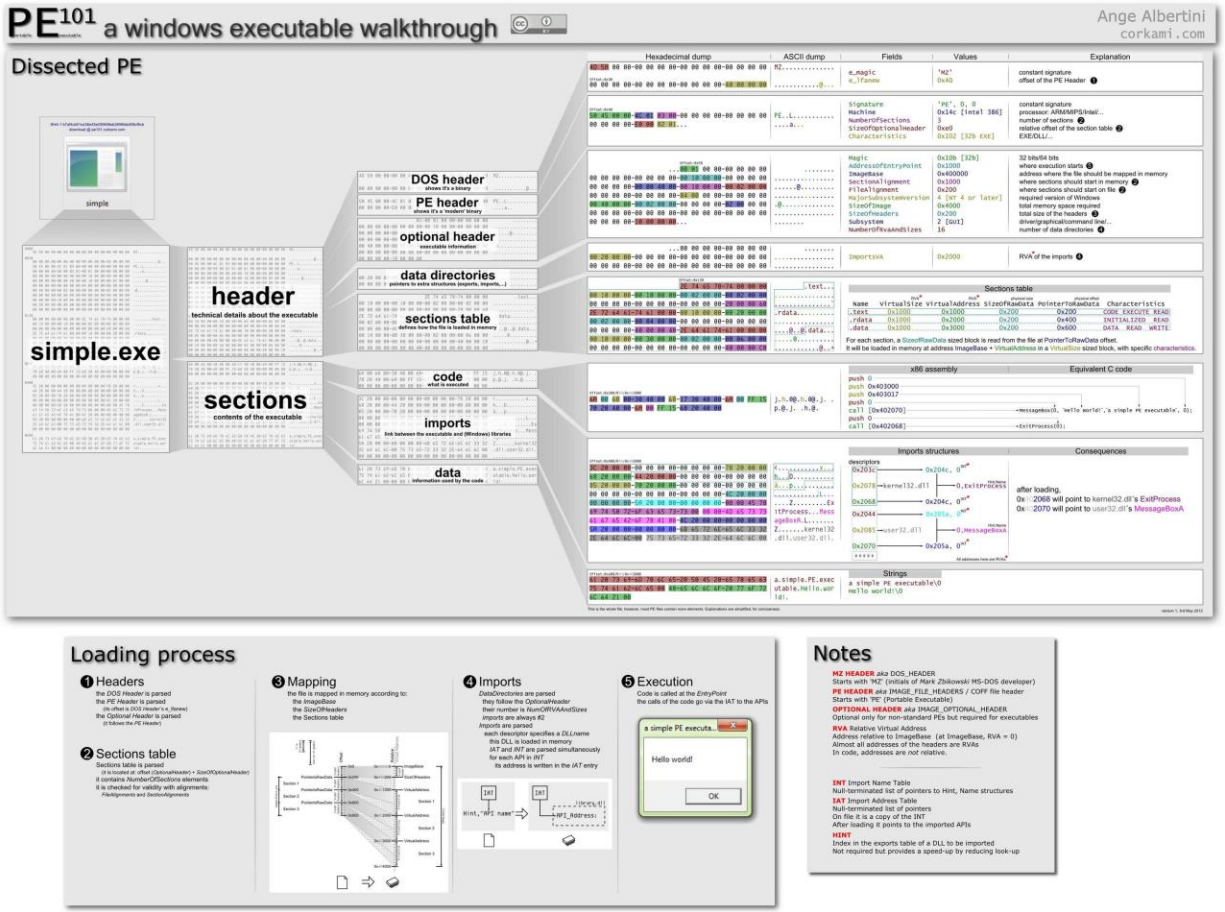
- This is how your rule should look like:

```
rule PE_file
{
        strings:
                $mz = "MZ"
                $pe = "PE"
        condition:
                ($mz at 0) and
                ($pe at (uint32(0x3c)))
}
```

PE format

- PE = Portable Executable
- Structured format for Windows executable files
- Supporting documents in /yara/doc/PE
  - Overview by Ange Albertini
  - Specification v8.3 by Microsoft (2013)

PE format


PE101 a windows executable walkthrough
Ange Albertini
corkami.com
Dissected PE
simple.exe
header
sections
DOS header
PE header
optional header
data directories
sections table
code
imports
data
Loading process
Headers
Sections table
Mapping
Imports
Execution
Notes

Identify Other Files

- Write a rule to detect a PDF file
- Write a rule to detect a ZIP file

- What is needed?

Identify Other Files

- Write a rule to detect a PDF file
- Write a rule to detect a ZIP file

- What is needed?
  - This is what you need: List

Detecting PDF File Type

- Write a rule to detect the PDF type

```
rule DetectPDF
{
    strings:
        $pdf = { 25 50 44 46 } // %PDF
    condition:
        $pdf at 0
}
```

Checking Specific Values at a Location

- The malware version is at byte 0x20:

```
rule VersionCheck
{
    strings:
        $ver = "v1.2.3"
    condition:
        $ver at 0x20
}
```

Sets of Strings (Multiple Indicators)

- Match at least two indicators:

```
rule MultipleIndicators
{
    strings:
        $a = "keylogger"
        $b = "stealer"
        $c = "injector"
    condition:
        2 of ($a,$b,$c)
}
```

Using Regular Expressions

- Detecting data exfiltration via email :

```
rule EmailPattern
{
    strings:
        $email = /[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}/
    condition:
        $email
}
```

Matching on Machine Code

Matching Assembly Code Patterns

- Check code patterns: Detecting known shellcode patterns

```
rule SuspiciousASM
{
    strings:
        $asm = { 8B ?? ?? 89 ?? ?? 8D }
    condition:
        $asm
}
```

- 8B = MOV, 8D = LEA (common in shellcode)

About the malware samples

- Objective
  - How to build binary signatures that match on x86 machine code
- Sample: 44efa4accc42aa55d7843ec69161c8ca:
  - The sample is a backdoor belonging to the Hoardy family.

The first decryption routine

- Disassembly of sample 44efa4accc42aa55d7843ec69161c8ca:

```
.text:00401723                          decrypt:
.text:00401723 89 45 E8                 mov     [ebp+0BB4h+var_BCC], eax
.text:00401726 3B C7                    cmp     eax, edi
.text:00401728 7D 18                    jge     short end
.text:0040172A 8A 88 F0 E8 40 00        mov     cl, buffer[eax]
.text:00401730 32 C8                    xor     cl, al
.text:00401732 2A C8                    sub     cl, al
.text:00401734 80 E9 5A                 sub     cl, 5Ah
.text:00401737 88 88 F0 E8 40 00        mov     buffer[eax], cl
.text:0040173D 83 C0 01                 add     eax, 1
.text:00401740 EB E1                    jmp     short decrypt
```

The first decryption routine

- Create a rule file named “hoardy.yara”.
- Create a YARA rule which matches on the bytes that are typeset in bold letters (see previous page).
- Name your rule “crypto1” and tag it as “category”.
- Name the string “$crypto1”, too.
- Try your rule on the sample.

Practice Time!

- Get one c code executable from the previous labs
- Analyze the executable file
- Write a Yara rule to match a function pattern
- Where to find the instructions?
  - X86 architecture: http://ref.x86asm.net/coder32.html

Conclusion

Conclusion

Strings

- Text
  - make use of modifiers: nocase, fullword, ascii, wide
- Hex
  - make use of wildcards and jumps
- Perl compatible regular expressions

Conclusion

Condition

- Sets
  - 2 of ($a,$b,$c)
  - any of them
  - all of them
- Count number of string matches: #string
- Iterator “for”
- Offsets:
  - at offset
  - Entrypoint
- Access raw bytes: int8..int32, uint8..uint32
- Keep your rules simple, reference other rules

Conclusion

Metadata

- Define metadata
  - string
  - integer
  - boolean
- Examples:
  - weight (count of matching bits)
  - Architecture
  - Algorithm
  - endianess
- Use “-m” command line option to display metadata

Conclusion

Maintain a set of rules

- One-file-to-keep-them-all doesn't work well
- Refactor your rules
  - write rules for each common expression ("primitives")
  - separate files by topic, make use of "include"
- Rule modifiers:
  - "global" makes rule a prerequisite for all other rules (e.g. PE header check)
  - "private" suppresses output
- Make use of tags and "-t" command line option to select rules

Conclusion

More information

- YARA manuals and wiki at
  - http://code.google.com/p/yara-project/

Conclusion

- Format of Yara rule
- Write Yara rule
- PE format
- Write rule for binary matching

Reference

- Some slides' content are borrowed from Andreas Schuster presentation slide on 26th FIRST conference in Boston, 2014:
  - https://www.first.org/resources/papers/conference2014/first_2014_-_schuster-_andreas_-_yara_basic_and_advanced_20140619.pdf

- The slides are changed to match the requirement of the class