YARA User's Manual Ver. 1.1

Víctor Manuel Álvarez
vmalvarez@hispasec.com

1. YARA in a nutshell

YARA is a tool aimed at helping malware researchers to identify and classify malware families. With YARA you can create descriptions of malware families based on textual or binary information contained on samples of those families. These descriptions, named rules, consist of a set of strings and a Boolean expression which determines the rule logic.

Let's explain it with an example. Suppose that we have a malware family with two variants, one of them downloads a malicious file from *http://foo.com/badfile1.exe*, the other downloads a file from *http://bar.com/badfile2.exe*, the URLs are hardcoded into the malware code. Both variants drops the downloaded file with the name *win.exe*, which also appears hardcoded into the samples. For this hypothetical family we can create a rule like this:

```
rule BadBoy
{
      strings:
         $a = "win.exe"
         $b = "http://foo.com/badfile1.exe"
         $c = "http://bar.com/badfile2.exe"

      condition:
         $a and ($b or $c)
}
```

The rule above instructs YARA that those files containing the string *win.exe* and any of the two URLs must be reported as BadBoy.

This is just a simple example, but more complex and powerful rules can be created by using binary strings with wild-cards, case-insensitive text strings, regular expressions, and many other features provided by YARA that will be covered in this manual.

2. Writing rules

YARA rules are easy to write and understand, and they have a syntax that resembles in some way a C struct declaration. He here is the simplest rule that you can write for YARA, which does absolutely nothing:

```
rule Dummy
{
      condition:
         false
}
```

Each rule in YARA starts with the keyword `rule` followed by a rule identifier. Identifiers must follow the same lexical conventions of the C programming language, they can contain any alphanumeric character and the underscore character, but the first character can not be a digit. Rule identifiers are case sensitive and cannot exceed 128 characters. The following keywords are reserved and cannot be used as an identifier:

```
and                     or
at                      of
condition               private
entrypoint              rule
false                   rva
filesize                section
fullword                strings
is                      them
nocase                  true
not                     widechar
```

Rules are generally composed of two sections: strings definition and condition, although the strings definition section can be omitted if the rule doesn't rely on any string. The condition section is always required. The strings definition section is where the strings that will be part of the rule are defined. Each string has an identifier consisting in a $ character followed by a sequence of alphanumeric characters and underscores, these identifiers can be used in the condition section to refer to the corresponding string. Strings can be defined in text or hexadecimal form, as shown in the following example:

```
rule ExampleRule
{
    strings:
        $my_text_string = "text here"
        $my_hex_string = { E2 34 A1 C8 23 FB }

    condition:
        $my_text_string or $my_hex_string
}
```

Text strings are enclosed on double quotes just like in the C language. Hex strings are enclosed by curly brackets, and they are composed by a sequence of hexadecimal numbers that can appear contiguously or separated by spaces. Decimal numbers are not allowed in hex strings.

The condition section is where the logic of the rule resides. This section must contain a Boolean expression telling under which circumstances a file satisfies the rule or not. Generally, the condition will refer to previously defined strings by using the string identifier. In this context the string identifier acts as a Boolean variable which evaluate to true of the string was found in the file, or false otherwise. If the condition is true for a given file, the file matches the rule.

3. Strings

There are three types of strings in YARA: hexadecimal strings, text strings and regular expressions. Hexadecimal strings are used for defining raw sequences of bytes, while text strings and regular expressions are useful for defining portions of legible text. However text strings and regular expressions can be also used for representing raw bytes by mean of escape sequences as will be shown below.

3.1 Hexadecimal strings

Hexadecimal strings allow two special constructions that make them more flexible: wild-cards and jumps. Wild-cards are just placeholders that you can put into the string indicating that some bytes are unknown and they should match anything. The placeholder character is the question mark (?). Here you have an example of a hexadecimal string with wild-cards:

```
rule WildcardExample
{
      strings:
         $hex_string = { E2 34 ?? C8 A? FB }

      condition:
         $hex_string
}
```

As shown in the example the wild-cards are nibble-wise, which means that you can define just one nibble of the byte and leave the other unknown.

Wild-cards are useful when defining strings whose content can vary but you know the length of the variable chunks, however, this is not always the case. In some circumstances you may need to define strings with chunks of variable content and length. In those situations you can use jumps instead of wild-cards.

```
rule JumpExample
{
      strings:
         $hex_string = { F4 23 [4-6] 62 B4 }

      condition:
         $hex_string
}
```

In the example above we have a pair of numbers enclosed in square brackets and separated by a hyphen, that's a jump. This jump is indicating that any arbitrary sequence from 4 to 6 bytes can occupy the position of the jump. Any of the following strings will match the pattern:

```
F4 23 01 02 03 04 62 B4
F4 23 00 00 00 00 00 62 B4
F4 23 15 82 A3 04 45 22 62 B4
```

The lower bound of a jump (the number before the hyphen) must be greater than or equal to zero and lower than the higher bound (the number after the hyphen), while the higher bound cannot exceed 255. These are valid jumps:

```
FE 39 45 [0-8] 89 00
FE 39 45 [23-45] 89 00
FE 39 45 [128-255] 89 00
```

These are invalid:

```
FE 39 45 [10-7] 89 00
FE 39 45 [4-4] 89 00
FE 39 45 [200-300] 89 00
```

A jump can be also specified by a single number enclosed in brackets like this:

```
FE 39 45 [6] 89 00
```

That means that exactly 6 bytes must exist in the place occupied by the jump, and it's equivalent to:

```
FE 39 45 ?? ?? ?? ?? ?? ?? 89 00
```

Of course, wild-cards and jumps can be mixed together in the same string:

```
78 [4] 45 ?? 5F
```

The only limitation to wild-cards and jumps is that they can not appear at the beginning of the string. The follow strings are invalid:

```
5? 00 40 23
[4] 12 35
```

3.2 Text strings

As shown in previous sections, text strings are generally defined like this:

```
rule TextExample
{
      strings:
         $text_string = "foobar"

      condition:
         $text_string
}
```

This is the simplest case: an ASCII-encoded, case-sensitive string. However, text strings can be accompanied by some useful modifiers that alter the way in which the string will be interpreted. Those modifiers are appended at the end of the string definition separated by spaces, as will be discussed below.

Text strings can also contain the following subset of the escape sequences available in the C language:

`\"`	Double quote
`\\`	Backslash
`\t`	Horizontal tab
`\x`*dd*	Any byte in hexadecimal notation

3.2.1 Case-insensitive strings

Text strings in YARA are case-sensitive by default, however you can turn your string into case-insensitive mode by appending the modifier `nocase` at the end of the string definition, in the same line.

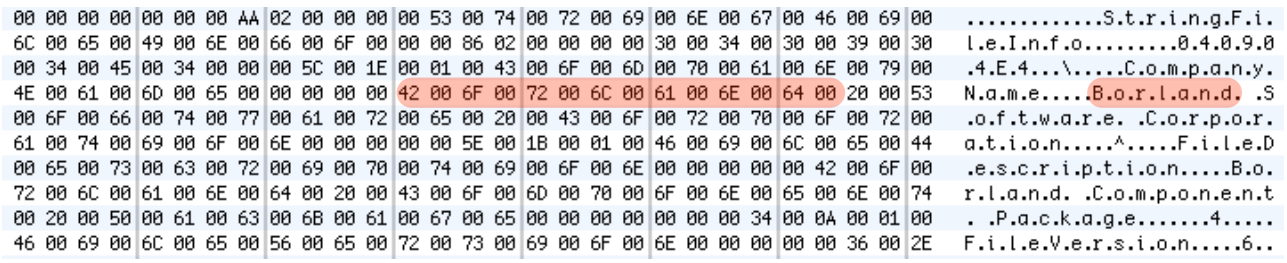
```
rule CaseInsensitveTextExample
{
      strings:
         $text_string = "foobar" nocase

      condition:
         $text_string
}
```

With the `nocase` modifier the string "foobar" will match "Foobar", "FOOBAR", and "fOoBaR".

This modifier can be used in conjunction with any other modifier.

3.2.2 Wide-character strings

The `widechar` modifier can be used to search for strings encoded with two bytes per character, something typical in many executable binaries.



In the above figure de string "Borland" appears encoded as two bytes per character, therefore the following rule will match:

```
rule WideCharTextExample
{
      strings:
         $wide_string = "Borland" widechar

      condition:
         $wide_string
}
```

However, keep in mind that this modifier just interleaves the ASCII codes of the characters in the string with zeroes, it does not support truly UTF-16 strings containing non-English characters.

This modifier can be used in conjunction with any other modifier.

3.2.3 Searching for full words

Another modifier that can be applied to text strings is `fullword`. This modifier guarantee that the string will match only if it appears in the file delimited by non-alphanumeric characters. For example the string "domain", if defined as `fullword`, don't matches "www.mydomain.com" but it matches "www.my-domain.com" and "www.domain.com".

3.3 Regular expressions

Regular expressions are one of the most powerful features of YARA. They are defined in the same way as text strings, but enclosed in backslashes instead of double-quotes, like in the Perl programming language. The regular expression syntax is also Perl-compatible.

```
rule RegExpExample1
{
      strings:
         $re1 = /md5: [0-9a-zA-Z]{32}/
         $re2 = /state: (on|off)/

      condition:
         $re1 and $re2
}
```

Regular expressions can be also followed by `nocase`, `widechar` and `fullword` modifiers just like in text strings. The semantics of these modifiers are the same in both cases.

For more information about Perl regular expressions please visit:
http://www.pcre.org/pcre.txt

4. Conditions

Conditions are nothing more than Boolean expressions as those that can be found in all programming languages, for example in an "if" statement. They can contain the typical Boolean operators `and`, `or` and `not` and relational operators `>=`, `<=`, `<`, `>`, `==` and `!=`. Also, the arithmetic operators (`+`, `-`, `*`, `/`) can be used on numerical expressions.

String identifiers can be also used within a condition, acting as Boolean variables whose value depends on the presence or not of the associated string in the file.

```
rule Example
{
      strings:
         $a = "text1"
         $b = "text2"
         $c = "text3"
         $d = "text4"

      condition:
         ($a or $b) and ($c or $d)
}
```

4.1 Restricting string offsets

In the majority of cases, when a string identifier is used in a condition, we are willing to know if the associated string is anywhere within the file, but sometimes we need to know if the string is at some specific offset on the file. In such situations the operator `at` is what we need. This operator is used as shown in the following example:

```
rule AtExample
{
      strings:
         $a = "dummy1"
         $b = "dummy2"

      condition:
         $a at 100 and $b at 200
}
```

The expression `$a at 100` in the above example is true only if string `$a` is found at offset 100 within the file. The string `$b` should appear at offset 200. Please note that both offsets are decimal, however hexadecimal numbers can be written by adding the prefix 0x before the number as in the C language. Also note the higher precedence of the operator `at` over the `and`.

While the `at` operator allows to search for a string at some fixed offset in the file, the `in` operator allows to search for the string within a range of offsets:

```
rule InExample
{
      strings:
         $a = "dummy1"
         $b = "dummy2"

      condition:
         $a in [0..100] and $b in [100..filesize]
}
```

In the example above the string `$a` must be found at an offset between 0 and 100, while string `$b` must be at an offset between 100 and the end of the file. Again the numbers are decimal by default.

4.1 File size

String identifiers are not the only variables that can appear in a condition (in fact, rules can be defined without any string definition as will be shown below), there are other special variables that can be used as well. One of these especial variables is `filesize`, which holds, as its name indicates, the size of the file being analyzed. The size is expressed in bytes.

```
rule FileSizeExample
{
      condition:
         filesize > 200KB
}
```

The previous example also demonstrate the use of the KB postfix. This postfix, when attached to a numerical constant, automatically multiplies the value of the constant by 1024. The MB postfix can be used to multiply the value by 2^{20}. Both postfixes can be used only with decimal constants.

4.2 Entry point

Another special variable than can be used on a rule is `entrypoint`. If the file is a Portable Executable (PE), this variable holds the raw offset (not the RVA) of the entry point of the executable. A typical use of this variable is to look for some pattern at the entry point to detect packers or simple PE infectors.

```
rule EntryPointExample1
{
      strings:
         $a = { E8 00 00 00 00 }

      condition:
         $a at entrypoint
}
rule EntryPointExample2
```

```
{
      strings:
         $a = { 9C 50 66 A1 ?? ?? ?? 00 66 A9 ?? ?? 58 0F 85 }

      condition:
         $a in [entrypoint..entrypoint + 10]
}
```

The presence of the `entrypoint` variable in a rule implies that only PE files can satisfy that rule. If the file is not a PE any rule using this variable evaluates to false.

4.4 Counting strings

Sometimes we need to know not only if certain string is in the file or not, but how many times the string appears in the file. The number of occurrences of each string is represented by a variable whose name is the string identifier but with a # character in place of the $ character. For example:

```
rule CountExample
{
      strings:
         $a = "dummy1"
         $b = "dummy2"

      condition:
         #a == 6 and #b > 10
}
```

This rules match any file containing the string `$a` exactly six times, and more than ten occurrences of string `$b`.

4.5 Sets of expressions

There are circumstances in which is necessary to express that the file should comply with a certain number expressions from a given set. None of the expressions are required to be true, but at least some of them should be. In these situations the operator `of` come into help.

```
rule OfExample
{
      strings:
         $a = "dummy1"
         $b = "dummy2"
         $c = "dummy3"
         $d = "dummy4"

      condition:
         2 of ($a,$b at 300,#c > 5)
}
```

What this rule says is that at least two of the three expressions must be true, no matter which. Of course, when using this operator, the number before the `of` keyword must be equal to or less than the number of expressions that appear between the parenthesis.

You can also replace the list of expressions with the keyword `them`. This is a syntactic sugar equivalent to a list containing all the strings defined in the rule.

```
rule OfThemExample
{
      strings:
         $a = "dummy1"
         $b = "dummy2"
         $c = "dummy3"
         $d = "dummy4"

      condition:
         2 of them /* equivalent to 2 of ($a,$b,$c,$d) */
}
```

4.6 Referencing other rules

When writing the condition for a rule you can also make reference to a previously defined rule in a manner that resembles a function invocation of traditional programming languages. In this way you can create rules that depends on others. Let's see an example:

```
rule Rule1
{
      strings:
         $a = "dummy1"

      condition:
         $a
}

rule Rule2
{
      strings:
         $a = "dummy2"

      condition:
         $a and Rule1
}
```

As can be seen in the example, a file will satisfy Rule2 only if it contains the string "dummy2" and satisfy Rule1. Note that is strictly necessary to define the rule being invoked before the one that will make the invocation.

5. More about rules

There are some aspects of YARA rules that has not been covered yet, but still are very important. They are: private rules and rule tags.

5.1 Private rules

Private rules are a very simple concept. That are just rules that are not reported by YARA when they match on a given file. Rules that are not reported at all may seem sterile at first glance, but when mixed with the possibility offered by YARA of referencing one rule from another (see section 4.5) they become useful. Private rules can serve as building blocks for other rules, and at the same time prevent cluttering YARA's output with irrelevant information. For declaring a rule as private the keyword `private` must appear just before `rule`.

```
private rule PrivateRuleExample
{
    ...
}
```

5.2 Rule tags

Another useful feature of YARA is the possibility of adding tags to rules. Those tags can be used later to filter YARA's output and show only the rules that you are interesting in. You can add as many tags as you want to a rule, they are declared after the rule identifier as shown below:

```
rule TagsExample1 : Foo Bar Baz
{
    ...
}

rule TagsExample2 : Bar
{
    ...
}

rule TagsExample3 : Foo Baz
{
    ...
}
```

Tags must follow the same lexical convention of rule identifiers, therefore only alphanumeric characters and underscores are allowed, and the tag cannot start with a digit. They are also case sensitive.

When using YARA you can output only those rules that are tagged with the tag or tags that you provide. More details on this topic can be found in the following section.

6. Using YARA from command line

In order to invoke YARA you will need two things: the set of rules you want to apply, and the path to the file or folder that you want to scan. The rules can be provided to YARA through one or more plain-text files containing the rules, or through the standard input if no rule file is specified.

```
usage:  yara [ -t tag ] [ -g ] [ -s ] [ -r ] [ -n ] [ -v ] [RULEFILE...] FILE
options:
  -t <tag>          print rules tagged as <tag> and ignore the rest.
  -n                print rules that doesn't apply (negate).
  -g                print tags.
  -s                print strings.
  -r                recursively search directories.
  -v                show version information.
```

The rules will be applied to the file specified as the last argument to YARA, if this path points to a directory, all the files contained in it will be scanned. By default YARA does not attempt to scan directories recursively, but you can use the `-r` option to do it.

The `-t` option allows you to specify one or more tags that will act as filters to YARA's output. If you use this option, only those rules tagged as you specify will be shown. You can also use the `-n` modifier to print those rules that are not satisfied by the files.

7. Using YARA from Python

YARA can be also invoked from your own Python scripts. The `yara-python` extension is provided in order to make YARA functionality available to Python users. Once `yara-python` is built and installed on your system you can use it as shown below:

```
import yara
```

Then you will need to compile the YARA rules before applying them to your data:

```
rules = yara.compile('/foo/bar/myrules')
```

The method `compile` of this module returns an instance of the class `Rules`, which in turn has two methods: `matchfile` and `match`. The first one applies the rules to a file given its path:

```
matches = rules.matchfile('/foo/bar/myfile')
```

The second one applies the rules to a Python string:

```
f = fopen('/foo/bar/myfile', 'rb')

matches = rules.match(f.read())
```

Both methods return a list of instances of the class `Match`. The instances of this class can be treated as text strings containing the name of the matching rule. For example you can print them:

```
foreach m in matches:
    print "%s" % m
```

In some circumstances you may need to explicitly convert the instance of `Match` to string, for example when comparing it with another string:

```
if str(matches[0]) == 'SomeRuleName':
    ...
```

The `Match` class have another two attributes: `tags` and `strings`. The `tags` attribute is a list of strings containing the tags associated to the rule. The `strings` attribute is a dictionary whose values are those strings within the data that made the YARA rule match, and the keys are the offsets where those strings were found.