Writing YARA rules

YARA rules are easy to write and understand, and they have a syntax that resembles the C language. 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:

YARA keywords¶

all	and	any	ascii	at	condition	contains
entrypoint	false	filesize	fullword	for	global	in
import	include	int8	int16	int32	int8be	int16be
int32be	matches	meta	nocase	not	or	of
private	rule	strings	them	true	uint8	uint16
uint32	uint8be	uint16be	uint32be	wide		

Rules are generally composed of two sections: strings definition and condition. The strings definition section can be omitted if the rule doesn't rely on any string, but 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 or process satisfies the rule or not. Generally, the condition will refer to previously defined strings by using their identifiers. In this context the string identifier acts as a boolean variable which evaluate to true of the string was found in the file or process memory, or false if otherwise.

Comments

You can add comments to your YARA rules just as if it was a C source file, both single-line and multi-line C-style comments are supported.

```
/*
    This is a multi-line comment ...
*/

rule CommentExample   // ... and this is single-line comment
{
    condition:
       false  // just an dummy rule, don't do this
}
```

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.

Hexadecimal strings

Hexadecimal strings allow three special constructions that make them more flexible: wild-cards, jumps, and alternatives. 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
```

Any jump [X-Y] must met the condition 0 <= X <= Y. In previous versions of YARA both X and Y must be lower than 256, but starting with YARA 2.0 there is no limit for X and Y.

These are valid jumps:

```
FE 39 45 [0-8] 89 00
FE 39 45 [23-45] 89 00
FE 39 45 [1000-2000] 89 00
```

This is invalid:

```
FE 39 45 [10-7] 89 00
```

If the lower and higher bounds are equal you can write a single number enclosed in brackets, like this:

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

The above string is equivalent to both of these:

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

Starting with YARA 2.0 you can also use unbounded jumps:

```
FE 39 45 [10-] 89 00
FE 39 45 [-] 89 00
```

The first one means `[10-infinite]`, the second one means `[0-infinite]`.

There are also situations in which you may want to provide different alternatives for a given fragment of your hex string. In those situations you can use a syntax which resembles a regular expression:

```
rule AlternativesExample1
{
    strings:
       $hex_string = { F4 23 ( 62 B4 | 56 ) 45 }

    condition:
       $hex_string
}
```

This rule will match any file containing `F42362B445` or `F4235645`.

But more than two alternatives can be also expressed. In fact, there are no limits to the amount of alternative sequences you can provide, and neither to their lengths.

```
rule AlternativesExample2
{
    strings:
       $hex_string = { F4 23 ( 62 B4 | 56 | 45 ?? 67 ) 45 }

    condition:
       $hex_string
}
```

As can be seen also in the above example, strings containing wild-cards are allowed as part of alternative sequences.

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
`\n`	New line
`\xdd`	Any byte in hexadecimal notation

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.

Wide-character strings

The `wide` 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" wide

    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. If you want to search for strings in both ASCII and wide form, you can use the `ascii` modifier in conjunction with `wide`, no matter the order in which they appear.

```
rule WideCharTextExample
{
    strings:
        $wide_and_ascii_string = "Borland" wide ascii

    condition:
       $wide_and_ascii_string
}
```

The `ascii` modifier can appear along, without an accompanying `wide` modifier, but it's not necessary to write it because in absence of `wide` the string is assumed to be ASCII by default.

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*.

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.

```
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`, `ascii`, `wide`, and `fullword` modifiers just like in text strings. The semantics of these modifiers are the same in both cases.

In previous versions of YARA externals libraries like PCRE and RE2 were used to perform regular expression matching, but starting with version 2.0 YARA uses its own regular expression engine. This new engine implements most features found in PCRE, except a few of them like capture groups, POSIX character classes and backreferences.

YARA's regular expressions recognise the following metacharacters:

`\`	Quote the next metacharacter
`^`	Match the beginning of the file
`$`	Match the end of the file
`|`	Alternation
`()`	Grouping
`[]`	Bracketed character class

The following quantifiers are recognised as well:

`*`	Match 0 or more times
`+`	Match 1 or more times
`?`	Match 0 or 1 times
`{n}`	Match exactly n times
`{n,}`	Match at least n times
`{,m}`	Match 0 to m times

`{n,m}`	Match n to m times

All these quantifiers have a non-greedy variant, followed by a question mark (?):

`*?`	Match 0 or more times, non-greedy
`+?`	Match 1 or more times, non-greedy
`??`	Match 0 or 1 times, non-greedy
`{n}?`	Match exactly n times, non-greedy
`{n,}?`	Match at least n times, non-greedy
`{,m}?`	Match 0 to m times, non-greedy
`{n,m}?`	Match n to m times, non-greedy

The following escape sequences are recognised:

`\t`	Tab (HT, TAB)
`\n`	New line (LF, NL)
`\r`	Return (CR)
`\n`	New line (LF, NL)
`\f`	Form feed (FF)
`\a`	Alarm bell
`\xNN`	Character whose ordinal number is the given hexadecimal number

These are the recognised character classes:

`\w`	Match a *word* character (aphanumeric plus "_")
`\W`	Match a *non-word* character
`\s`	Match a whitespace character
`\S`	Match a non-whitespace character
`\d`	Match a decimal digit character
`\D`	Match a non-digit character

Starting with version 3.3.0 these zero-with assertions are also recognized:

`\b`	Match a word boundary
`\B`	Match except at a word boundary

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 (+, -, *, \, %) and bitwise 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)
}
```

Counting strings

Sometimes we need to know not only if a certain string is present or not, but how many times the string appears in the file or process memory. 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 or process containing the string $a exactly six times, and more than ten occurrences of string $b.

String offsets or virtual addresses

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 or process memory, but sometimes we need to know if the string is at some specific offset on the file or at some virtual address within the process address space. 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 (or at virtual address 100 if applied to a running process). 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, which comes very handy when writing virtual addresses. 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 or virtual address in a process memory space, the `in` operator allows to search for the string within a range of offsets or addresses.

```
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, numbers are decimal by default.

You can also get the offset or virtual address of the i-th occurrence of string $a by using @a[i]. The indexes are one-based, so the first occurrence would be @a[1] the second one @a[2] and so on. If you provide an index greater then the number of occurrences of the string, the result will be a NaN (Not A Number) value.

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 scanned. 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.

The use of `filesize` only makes sense when the rule is applied to a file, if the rule is applied to a running process won't never match because `filesize` doesn't make sense in this context.

Executable entry point

Another special variable than can be used on a rule is `entrypoint`. If file is a Portable Executable (PE) or Executable and Linkable Format (ELF), this variable holds the raw offset of the exectutable's entry point in case we scanning a file. If we are scanning a running process entrypoint will hold the virtual address of the main executable's entry point. A typical use of this variable is to look for some pattern at the entry point to detect packers or simple file 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 or ELF files can satisfy that rule. If the file is not a PE or ELF any rule using this variable evaluates to false.

Warning

The `entrypoint` variable is deprecated, you should use the equivalent `pe.entry_point` from the PE module instead. Starting with YARA 3.0 you'll get a warning if you use `entrypoint` and it will be completely removed in future versions.

Accessing data at a given position

There are many situations in which you may want to write conditions that depends on data stored at a certain file offset or memory virtual address, depending if we are scanning a file or a running process. In those situations you can use one of the following functions to read data from the file at the given offset:

```
int8(<offset or virtual address>)
int16(<offset or virtual address>)
int32(<offset or virtual address>)

uint8(<offset or virtual address>)
uint16(<offset or virtual address>)
uint32(<offset or virtual address>)

int8be(<offset or virtual address>)
int16be(<offset or virtual address>)
int32be(<offset or virtual address>)

uint8be(<offset or virtual address>)
uint16be(<offset or virtual address>)
uint32be(<offset or virtual address>)
```

The `intXX` functions read 8, 16, and 32 bits signed integers from <offset or virtual address>, while functions `uintXX` read unsigned integers. Both 16 and 32 bits integer are considered to be little-endian. If you want to read a big-endian integer use the corresponding function ending in `be`. The <offset or virtual address> parameter can be any expression returning an unsigned integer, including the return value of one the `uintXX` functions itself. As an example let's see a rule to distinguish PE files:

```
rule IsPE
{
  condition:
     // MZ signature at offset 0 and ...
     uint16(0) == 0x5A4D and
     // ... PE signature at offset stored in MZ header at 0x3C
     uint32(uint32(0x3C)) == 0x00004550
}
```

Sets of strings

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

```
rule OfExample1
{
    strings:
        $a = "dummy1"
        $b = "dummy2"
        $c = "dummy3"

    condition:
        2 of ($a,$b,$c)
}
```

What this rule says is that at least two of the strings in the set ($a,$b,$c) must be present on the file, 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 strings in the set.

The elements of the set can be explicitly enumerated like in the previous example, or can be specified by using wild cards. For example:

```
rule OfExample2
{
    strings:
        $foo1 = "foo1"
        $foo2 = "foo2"
        $foo3 = "foo3"

    condition:
        2 of ($foo*)  /* equivalent to 2 of ($foo1,$foo2,$foo3) */
}

rule OfExample3
{
    strings:
        $foo1 = "foo1"
        $foo2 = "foo2"

        $bar1 = "bar1"
        $bar2 = "bar2"

    condition:
        3 of ($foo*,$bar1,$bar2)
}
```

You can even use ($*) to refer to all the strings in your rule, or write the equivalent keyword `them` for more legibility.

```
rule OfExample4
{
    strings:
        $a = "dummy1"
        $b = "dummy2"
        $c = "dummy3"

    condition:
        1 of them /* equivalent to 1 of ($*) */
}
```

In all the above examples the number of strings have been specified by a numeric constant, but any expression returning a numeric value can be used. The keywords any and all can be used as well.

```
all of them       /* all strings in the rule */
any of them       /* any string in the rule */
all of ($a*)      /* all strings whose identifier starts by $a */
any of ($a,$b,$c) /* any of $a, $b or $c */
1 of ($*)         /* same that "any of them" */
```

Applying the same condition to many strings

There is another operator very similar to `of` but even more powerful, the `for..of` operator. The syntax is:

```
for expression of string_set : ( boolean_expression )
```

And its meaning is: from those strings in `string_set` at least `expression` of them must satisfy `boolean_expression`.

In other words: `boolean_expression` is evaluated for every string in `string_set` and must be at least `expression` of them returning True.

Of course, `boolean_expression` can be any boolean expression accepted in the condition section of a rule, except for one important detail: here you can (and should) use a dollar sign ($) as a place-holder for the string being evaluated. Take a look to the following expression:

```
for any of ($a,$b,$c) : ( $ at entrypoint  )
```

The $ symbol in the boolean expression is not tied to any particular string, it will be $a, and then $b, and then $c in the three successive evaluations of the expression.

Maybe you already realised that the `of` operator is an special case of `for..of`. The following expressions are the same:

```
any of ($a,$b,$c)
for any of ($a,$b,$c) : ( $ )
```

You can also employ the symbols # and @ to make reference to the number of occurrences and the first offset of each string respectively.

```
for all of them : ( # > 3 )
for all of ($a*) : ( @ > @b )
```

Using anonymous strings with `of` and `for..of`

When using the `of` and `for..of` operators followed by them, the identifier assigned to each string of the rule is usually superfluous. As we are not referencing any string individually we don't not need to provide a unique identifier for each of them. In those situations you can declare anonymous strings with identifiers consisting only in the $ character, as in the following example:

```
rule AnonymousStrings
{
    strings:
        $ = "dummy1"
        $ = "dummy2"

    condition:
        1 of them
}
```

Iterating over string occurrences

As seen in String offsets or virtual addresses, the offsets or virtual addresses where a given string appears within a file or process address space can be accessed by using the syntax: @a[i], where i is an index indicating which occurrence of the string $a are you referring to. (@a[1], @a[2],...).

Sometimes you will need to iterate over some of these offsets and guarantee they satisfy a given condition. For example:

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

    condition:
        for all i in (1,2,3) : (@a[i] + 10 == @b[i])
}
```

The previous rule tells that the first three occurrences of $b should be 10 bytes away from the first three occurrences of $a.

The same condition could be written also as:

```
for all i in (1..3) : (@a[i] + 10 == @b[i])
```

Notice that we're using a range (1..3) instead of enumerating the index values (1,2,3). Of course, we're not forced to use constants to specify range boundaries, we can use expressions as well like in the following example:

```
for all i in (1..#a) : (@a[i] < 100)
```

In this case we're iterating over every occurrence of $a (remember that #a represents the number of occurrences of $a). This rule is telling that every occurrence of $a should be within the first 100 bytes of the file.

In case you want to express that only some occurrences of the string should satisfy your condition, the same logic seen in the `for..of` operator applies here:

```
for any i in (1..#a): ( @a[i] < 100 )
for 2 i in (1..#a): ( @a[i] < 100 )
```

In resume, the syntax of this operator is:

```
for expression identifier in indexes : ( boolean_expression )
```

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.

More about rules

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

Global rules

Global rules give you the possibility of imposing restrictions in all your rules at once. For example, suppose that you want all your rules ignoring those files that exceed certain size limit, you could go rule by rule doing the required modifications to their conditions, or just write a global rule like this one:

```
global rule SizeLimit
{
    condition:
        filesize < 2MB
}
```

You can define as many global rules as you want, they will be evaluated before the rest of the rules, which in turn will be evaluated only of all global rules are satisfied.

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 Referencing other rules) 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 just add the keyword `private` before the rule declaration.

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

You can apply both `private` and `global` modifiers to a rule, resulting a global rule that does not get reported by YARA but must be satisfied.

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
{
    ...
}
```

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.

Metadata

Besides the string definition and condition sections, rules can also have a metadata section where you can put additional information about your rule. The metadata section is defined with the keyword `meta` and contains identifier/value pairs like in the following example:

```
rule MetadataExample
{
    meta:
        my_identifier_1 = "Some string data"
        my_identifier_2 = 24
        my_identifier_3 = true

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

    condition:
        $my_text_string or $my_hex_string
}
```

As can be seen in the example, metadata identifiers are always followed by an equal sign and the value assigned to them. The assigned values can be strings, integers, or one of the boolean values true or false. Note that identifier/value pairs defined in the metadata section can not be used in the condition section, their only purpose is to store additional information about the rule.

Using modules

Modules are extensions to YARA's core functionality. Some modules like the the PE module and the Cuckoo module are officially distributed with YARA and some of them can be created by third-parties or even by yourself as described in Writing your own modules.

The first step to use a module is importing it with the `import` statement. These statements must be placed outside any rule definition and followed by the module name enclosed in double-quotes. Like this:

```
import "pe"
import "cuckoo"
```

After importing the module you can make use of its features, always using `<module name>.` as a prefix to any variable, or function exported by the module. For example:

```
pe.entry_point == 0x1000
cuckoo.http_request(/someregexp/)
```

Modules often leave variables in undefined state, for example when the variable doesn't make sense in the current context (think of `pe.entry_point` while scanning a non-PE file). YARA handles undefined values in way that allows the rule to keep its meaningfulness. Take a look at this rule:

```
import "pe"

rule test
{
  strings:
    $a = "some string"
  condition:
    $a and pe.entry_point == 0x1000
}
```

If the scanned file is not a PE you wouldn't expect this rule matching the file, even if it contains the string, because **both** conditions (the presence of the string and the right value for the entry point) must be satisfied. However, if the condition is changed to:

```
$a or pe.entry_point == 0x1000
```

You would expect the rule matching in this case if the file contains the string, even if it isn't a PE file. That's exactly how YARA behaves. The logic is simple: any arithmetic, comparison, or boolean operation will result in an undefined value if one of its operands is undefined, except for *OR* operations where an undefined operand is interpreted as a False.

External variables

External variables allow you to define rules which depends on values provided from the outside. For example you can write the following rule:

```
rule ExternalVariableExample1
{
    condition:
       ext_var == 10
}
```

In this case `ext_var` is an external variable whose value is assigned at run-time (see `-d` option of command-line tool, and `externals` parameter of `compile` and `match` methods in yara-python). External variables could be of types: integer, string or boolean; their type depends on the value assigned to them. An integer variable can substitute any integer constant in the condition and boolean variables can occupy the place of boolean expressions. For example:

```
rule ExternalVariableExample2
{
    condition:
       bool_ext_var or filesize < int_ext_var
}
```

External variables of type string can be used with operators contains and matches. The `contains` operator returns true if the string contains the specified substring. The operator `matches` returns true if the string matches the given regular expression.

```
rule ExternalVariableExample3
{
    condition:
        string_ext_var contains "text"
}

rule ExternalVariableExample4
{
    condition:
        string_ext_var matches /[a-z]+/
}
```

You can use regular expression modifiers along with the `matches` operator, for example, if you want the regular expression from the previous example to be case insensitive you can use `/[a-z]+/i`. Notice the `i` following the regular expression in a Perl-like manner. You can also use the `s` modifier for single-line mode, in this mode the dot matches all characters including line breaks. Of course both modifiers can be used simultaneously, like in the following example:

```
rule ExternalVariableExample5
{
    condition:
        /* case insensitive single-line mode */
        string_ext_var matches /[a-z]+/is
}
```

Keep in mind that every external variable used in your rules must be defined at run-time, either by using the `-d` option of the command-line tool, or by providing the `externals` parameter to the appropriate method in `yara-python`.

Including files

In order to allow you a more flexible organization of your rules files, YARA provides the `include` directive. This directive works in a similar way to the *#include* pre-procesor directive in your C programs, which inserts the content of the specified source file into the current file during compilation. The following example will include the content of *other.yar* into the current file:

```
include "other.yar"
```

The base path when searching for a file in an `include` directive will be the directory where the current file resides. For that reason, the file *other.yar* in the previous example should be located in the same directory of the current file. However you can also specify relative paths like these ones:

```
include "./includes/other.yar"
include "../includes/other.yar"
```

And you can also use absolute paths:

```
include "/home/plusvic/yara/includes/other.yar"
```

In Windows both slashes and backslashes are accepted, and don't forget to write the drive letter:

```
include "c:/yara/includes/other.yar"
include "c:\\yara\\includes\\other.yar"
```