yara Documentation

Release 2.2

Victor M. Alvarez

August 21, 2014

Contents

YARA is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples. With YARA you can create descriptions of malware families (or whatever you want to describe) based on textual or binary patterns. Each description, a.k.a rule, consists of a set of strings and a boolean expression which determine its logic. Let's see an example:

```
rule silent_banker : banker
{
    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 = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
    condition:
        $a or $b or $c
}
```

The above rule is telling YARA that any file containing one of the three strings must be reported as silent_banker. This is just a simple example, more complex and powerful rules can be created by using wild-cards, case-insensitive strings, regular expressions, special operators and many other features that you'll find explained in this documentation.

Contents:

Getting started

YARA is a multi-platform program running on Windows, Linux and Mac OS X. You can find the latest release at https://github.com/plusvic/yara/releases.

1.1 Compiling and installing YARA

Download the source tarball and get prepared for compiling it:

```
tar -zxf yara-3.0.0.tar.gz
cd yara-3.0.0
./bootstrap.sh
```

YARA uses GNU autotools, so it's compiled and installed in the standard way:

```
./configure
make
sudo make install
```

The Cuckoo module is not compiled into YARA by default, if you plan to use the Cuckoo module you must pass the `--enable-cuckoo` argument to the `configure` script. The Cuckoo module depends on the Jansson library, you'll need to install it beforehand. Some Debian and Ubuntu versions already include a package named `libjansson-dev`, if `sudo apt-get install libjansson-dev` doesn't work for you then get the source code from its repository.

To build and install the `yara-python` extension:

```
cd yara-python
python setup.py build
sudo setup.py install
```

Note: You may need to install the Python development package (usually `python-dev`) before compiling `yara-python`. Additionally, `yara-python` depends on the `libyara` library which gets installed with YARA, so don't proceed to build `yara-python` without previously installing YARA as described above.

1.2 Running YARA for the first time

Now that you have installed YARA you can write a very simple rule and use the command-line tool to scan some file:

```
echo "rule dummy { condition: true }" > my_first_rule
yara my_first_rule my_first_rule
```

Don't get confused by the repeated `my_first_rule` in the arguments to `yara`, I'm just passing the same file as both the rules and the file to be scanned. You can pass any file you want to be scanned (second argument).

If everything goes fine you should get the following output:

```
dummy my_first_rule
```

Which means that the file `my_first_rule` is matching the rule named `dummy`.

If you get an error like this:

```
yara: error while loading shared libraries: libyara.so.2: cannot open shared
object file: No such file or directory
```

It means that the loader is not finding the `libyara` library which is located in `/usr/local/lib`. In some Linux flavors the loader doesn't look for libraries in this path by default, we must instruct him to do so by adding `/usr/local/lib` to the loader configuration file `/etc/ld.so.conf`:

```
sudo echo "/usr/local/lib" >> /etc/ld.so.conf
sudo ldconfig
```

CHAPTER 2

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:

Table 2.1: YARA keywords

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

2.1 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
}
```

2.2 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.

2.2.1 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.

2.2.2 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
`\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*.

2.2.3 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.

```
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

2.3 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)
}
```

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.3.4 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 2.2 you'll get a warning if you use `entrypoint` and it will be completely removed in future versions.

2.3.5 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 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>)
```

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. 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
}
```

2.3.6 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" */
```

2.3.7 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 )
```

2.3.8 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
}
```

2.3.9 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 )
```

2.3.10 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.

2.4 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.

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.4.4 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.

2.5 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.

2.6 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`.

2.7 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"
```

CHAPTER 3

Modules

Modules are the way YARA provides for extending its features. They allow to define data structures and functions which can be used in your rules to express more complex conditions. Here you'll find described some modules officially distributed with YARA, but you can also learn how to write your own modules in the *Writing your own modules* section.

3.1 PE module

The PE module allows you to create more fine-grained rules for PE files by using attributes and features of the PE file format. This module exposes most of the fields present in a PE header and provides functions which can be used to write more expressive and targeted rules. Let's see some examples:

```
import "pe"

rule single_section
{
    condition:
        pe.number_of_sections == 1
}

rule control_panel_applet
{
    condition:
        pe.exports("CPlApplet")
}

rule is_dll
{
    condition:
        pe.characteristics & pe.DLL
}
```

3.1.1 PE module reference

machine

Integer with one of the following values:

MACHINE_I386

MACHINE_AMD64

Example: pe.machine == pe.MACHINE_AMD64

`subsystem`
Integer with one of the following values:

`SUBSYSTEM_UNKNOWN`

`SUBSYSTEM_NATIVE`

`SUBSYSTEM_WINDOWS_GUI`

`SUBSYSTEM_WINDOWS_CUI`

`SUBSYSTEM_OS2_CUI`

`SUBSYSTEM_POSIX_CUI`

`SUBSYSTEM_NATIVE_WINDOWS`

Example: pe.subsystem == pe.SUBSYSTEM_NATIVE

`timestamp`
PE timestamp.

`entry_point`
Entry point raw offset or virtual address depending if YARA is scanning a file or process memory respectively. This is equivalent to the deprecated `entrypoint` keyword.

`image_base`
Image base relative virtual address.

`characteristics`
Bitmap with PE characteristics. Individual characteristics can be inspected by performing a bitwise AND operation with the following constants:

`RELOCS_STRIPPED`

`EXECUTABLE_IMAGE`

`LINE_NUMS_STRIPPED`

`LOCAL_SYMS_STRIPPED`

`AGGRESIVE_WS_TRIM`

`LARGE_ADDRESS_AWARE`

`BYTES_REVERSED_LO`

`32BIT_MACHINE`

`DEBUG_STRIPPED`

`REMOVABLE_RUN_FROM_SWAP`

`NET_RUN_FROM_SWAP`

`SYSTEM`

`DLL`

`UP_SYSTEM_ONLY`

`BYTES_REVERSED_HI`

Example: pe.characteristics & pe.DLL

`linker_version`
An object with two integer attributes, one for each major and minor linker version.

major
Major linker version.

minor
Minor linker version.

os_version
An object with two integer attributes, one for each major and minor OS version.

major
Major OS version.

minor
Minor OS version.

image_version
An object with two integer attributes, one for each major and minor image version.

major
Major image version.

minor
Minor image version.

subsystem_version
An object with two integer attributes, one for each major and minor subsystem version.

major
Major subsystem version.

minor
Minor subsystem version.

number_of_sections
Number of sections in the PE.

sections
An zero-based array of section objects, one for each section the PE has. Individual sections can be accessed by using the [] operator. Each section object has the following attributes:

name
Section name.

characteristics
Section characteristics.

virtual_address
Section virtual address.

virtual_size
Section virtual size.

raw_data_offset
Section raw offset.

raw_data_size
Section raw size.

Example: pe.sections[0].name == ".text"

exports(function_name)
Function returning true if the PE exports *function_name* or false otherwise.

Example: pe.exports("CPlApplet")

imports(dll_name, function_name)
Function returning true if the PE imports *function_name* from *dll_name*, or false otherwise. *dll_name* is case insensitive.

Example: pe.imports("kernel32.dll", "WriteProcessMemory")

3.2 Cuckoo module

The Cuckoo module enables you to create YARA rules based on behavioral information generated by a Cuckoo sandbox. While scanning a PE file with YARA, you can pass additional information about its behavior to the `cuckoo` module and create rules based not only in what it *contains*, but also in what it *does*.

Suppose that you're interested in executable files sending a HTTP request to http://someone.doingevil.com. In previous versions of YARA you had to settle with:

```
rule evil_doer
{
    strings:
        $evil_domain = "http://someone.doingevil.com"

    condition:
        $evil_domain
}
```

The problem with this rule is that the domain name could be contained in the file for perfectly valid reasons not related with sending HTTP requests to http://someone.doingevil.com. Furthermore, the malicious executable could contain the domain name ciphered or obfuscated, in which case your rule would be completely useless.

But now with the `cuckoo` module you can take the behavior report generated for the executable file by your Cuckoo sandbox, pass it alongside the executable file to YARA, and write a rule like this:

```
import "cuckoo"

rule evil_doer
{
    condition:
        cuckoo.network.http_request(/http://someone\.doingevil\.com/)
}
```

Of course you can mix your behavior-related conditions with good old string-based conditions:

```
import "cuckoo"

rule evil_doer
{
    strings:
        $some_string = { 01 02 03 04 05 06 }

    condition:
        $some_string and
        cuckoo.network.http_request(/http://someone\.doingevil\.com/)
}
```

But how do we pass the behavior information to the `cuckoo` module? Well, in the case of the command-line tool you must use the `-x` option in this way:

```
$yara -x cuckoo=behavior_report_file rules_file pe_file
```

`behavior_report_file` is the path to a file containing the behavior file generated by the Cuckoo sandbox in JSON format.

If you are using `yara-python` then you must pass the behavior report in the `modules_data` argument for the `match` method:

```
import yara
rules = yara.compile('./rules_file')
report_file = open('./behavior_report_file')
report_data = report_file.read()
rules.match(pe_file, modules_data={'cuckoo': bytes(report_data)})
```

Important: The `cuckoo` module is not built into YARA by default, to learn how to build YARA with Cuckoo support refer to *Compiling and installing YARA*.

3.2.1 Cuckoo module reference

`network`

`http_request` (regexp)
Function returning true if the program sent a HTTP request to a URL matching the provided regular expression.

Example: cuckoo.network.http_request(/evil\.com/)

`http_get` (regexp)
Similar to `http_request()`, but only takes into account GET requests.

`http_post` (regexp)
Similar to `http_request()`, but only takes into account POST requests.

`registry`

`key_access` (regexp)
Function returning true if the program accessed a registry entry matching the provided regular expression.

Example: cuckoo.registry.key_access(/\\Software\\Microsoft\\Windows\\CurrentVersion\\Run/)

`filesystem`

`file_access` (regexp)
Function returning true if the program accessed a file matching the provided regular expression.

Example: cuckoo.filesystem.file_access(/autoexec\.bat/)

`sync`

`mutex` (regexp)
Function returning true if the program opens or create a mutex matching the provided regular expression.

Example: cuckoo.sync.mutex(/EvilMutexName/)

CHAPTER 4

Writing your own modules

Starting with YARA 3.0 you can extend its features by using modules. With modules you can define data structures and functions which can be later used from your rules to express more complex and refined conditions. You can see some examples of what a module can do in the *Using modules* section.

The purpose of this sections is teaching you how to create your own modules for giving YARA that cool feature you always dreamed of.

4.1 The "Hello World!" module

Modules are written in C and built into YARA as part of the compiling process. In order to create your own modules you must be familiarized with the C programming language and how to configure and build YARA from source code. You don't need to understand how YARA does its magic, YARA exposes a simple API for modules which is all you'll need to know.

The source code for your module must reside in the *libyara/modules* directory in the source tree. It's recommended to use the module name as the file name for the source file, if your module's name is *foo* its source file should be *foo.c*.

In the *libyara/modules* directory you'll find a *demo.c* file which we'll use as our starting point. The file looks like this:

```
#include <yara/modules.h>

#define MODULE_NAME demo

begin_declarations;

  declare_string("greeting");

end_declarations;

int module_load(
    YR_SCAN_CONTEXT* context,
    YR_OBJECT* module,
    void* module_data,
    size_t module_data_size)
{
  set_string("Hello World!", module, "greeting");
  return ERROR_SUCCESS;
}

int module_unload(
    YR_OBJECT* module)
```

```
{
  return ERROR_SUCCESS;
}

#undef MODULE_NAME
```

Let's start dissecting the source code so you can understand every detail. The first line in the code is:

```
#include <yara/modules.h>
```

The *modules.h* header file is where the definitions for YARA's module API reside, therefore this include directive is required in all your modules. The second line is:

```
#define MODULE_NAME demo
```

This is how you define the name of your module and is also required. Every module must define its name at the start of the source code. Module names must be unique among the modules built into YARA.

Then follows the declaration section:

```
begin_declarations;

  declare_string("greeting");

end_declarations;
```

Here is where the module declares the functions and data structures that will be available for your YARA rules. In this case we are declaring just a string variable named *greeting*. We are going to discuss more in depth about this in *The declaration section*.

Then comes the `module_load` function:

```
int module_load(
    YR_SCAN_CONTEXT* context,
    YR_OBJECT* module,
    void* module_data,
    size_t module_data_size)
{
  set_string("Hello World!", module, "greeting");
  return ERROR_SUCCESS;
}
```

This function is invoked once for each scanned file, but only if the module is imported by some rule with the `import` directive. The `module_load` function is where your module has the opportunity to inspect the file being scanned, parse it or analize it the way it may prefer, and then populate the data structures defined in the declarations section.

In this example the `module_load` function doesn't inspect the file content at all, it just assign the string "Hello World!" to the variable *greeting* declared before.

And finally we have the `module_unload` function:

```
int module_unload(
    YR_OBJECT* module)
{
  return ERROR_SUCCESS;
}
```

For each call to `module_load` there is a corresponding call to `module_unload`. This function allows your module to free any resource allocated during `module_load`. There's nothing to free in this case, so the function just returns `ERROR_SUCCESS`. Both `module_load` and `module_unload` should return `ERROR_SUCCESS` to indicate that everything went fine. If a different value is returned the scanning will be aborted and the error reported to the user.

4.1.1 Building our “Hello World!”

Modules are not magically built into YARA just by dropping their source code into the *libyara/modules* directory, you must follow two further steps in order to get them to work. The first step is adding your module to the *module_list* file also found in the *libyara/modules* directory.

The *module_list* file looks like this:

```
MODULE(tests)
MODULE(pe)

#ifdef CUCKOO
MODULE(cuckoo)
#endif
```

You must add a line *MODULE(<name>)* with the name of your module to this file. In our case the resulting *module_list* is:

```
MODULE(tests)
MODULE(pe)

#ifdef CUCKOO
MODULE(cuckoo)
#endif

MODULE(demo)
```

The second step is modifying the *Makefile.am* to tell the *make* program that the source code for your module most be compiled and linked into YARA. At the very beginning of *libyara/Makefile.am* you’ll find this:

```
MODULES =  modules/tests.c
MODULES += modules/pe.c

if CUCKOO
MODULES += modules/cuckoo.c
endif
```

Just add a new line for your module:

```
MODULES =  modules/tests.c
MODULES += modules/pe.c

if CUCKOO
MODULES += modules/cuckoo.c
endif

MODULES += modules/demo.c
```

And that’s all! Now you’re ready to build YARA with your brand-new module included. Just go to the source tree root directory and type as always:

```
make
sudo make install
```

Now you should be able to create a rule like this:

```
import "demo"

rule HelloWorld
{
    condition:
        demo.greeting == "Hello World!"
}
```

Any file scanned with this rule will match the `HelloWord` because `demo.greeting == "Hello World!"` is always true.

4.2 The declaration section

The declaration section is where you declare the variables, structures and functions that will be available for your YARA rules. Every module must contain a declaration section like this:

```
begin_declarations;

    <your declarations here>

end_declarations;
```

4.2.1 Basic types

Within the declaration section you can use `declare_string(<variable name>)` and `declare_integer(<variable name>)` to declare string or integer variables respectively. For example:

```
begin_declarations;

    declare_integer("foo");
    declare_string("bar");

end_declarations;
```

Variable names can't contain characters other than letters, numbers and underscores. These variables can be used later in your rules at any place where an integer or string is expected. Supposing your module name is "mymodule", they can be used like this:

```
mymodule.foo > 5

mymodule.bar matches /someregexp/
```

4.2.2 Structures

Your declarations can be organized in a more structured way:

```
begin_declarations;

    declare_integer("foo");
    declare_string("bar");

    begin_struct("some_structure");

        declare_integer("foo");

        begin_struct("nested_structure");
```

```
            declare_integer("bar");

        end_struct("nested_structure");

    end_struct("some_structure");

    begin_struct("another_structure");

        declare_integer("foo");
        declare_string("bar");
        declare_string("baz")

    end_struct("another_structure");

end_declarations;
```

In this example we're using `begin_struct(<structure name>)` and `end_struct(<structure name>)` to delimite two structures named *some_structure* and *another_structure*. Within the structure delimiters you can put any other declarations you want, including another structure declaration. Also notice that members of different structures can have the same name, but members within the same structure must have unique names.

When refering to these variables from your rules it would be like this:

```
mymodule.foo
mymodule.some_structure.foo
mymodule.some_structure.nested_structure.bar
mymodule.another_structure.baz
```

4.2.3 Arrays

In the same way you declare individual strings, integers or structures, you can declare arrays of them:

```
begin_declarations;

    declare_integer_array("foo");
    declare_string_array("bar");

    begin_struct_array("struct_array");

        declare_integer("baz");
        declare_string("qux");

    end_struct_array("struct_array");

end_declarations;
```

Individual values in the array are referenced like in most programming languages:

```
foo[0]
bar[1]
struct_array[3].baz
struct_array[1].qux
```

Arrays are zero-based and don't have a fixed size, they will grow as needed when you start initializing its values.

4.2.4 Functions

One of the more powerful features of YARA modules is the possibility of declaring functions that can be later invoked from your rules. Functions must appear in the declaration section in this way:

```
declare_function(<function name>, <argument types>, <return tuype>, <C function>);
```

<function name> is the name that will be used in your YARA rules to invoke the function.

<argument types> is a string containing one character per function argument, where the character indicates the type of the argument. Functions can receive three different types of arguments: string, integer and regular expression, denoted by characters: **s**, **i** and **r** respectively. If your function receives two integers *<argument types>* must be *"ii"*, if it receives an integer as the first argument and a string as the second one *<argument types>* must be *"is"*, if it receives three strings *<argument types>* must be "*sss*".

<return type> is a string with a single character indicating the return type. Possible return types are string (*"s"*) and integer (*"i"*).

<C function> is the identifier for the actual implementation of your function.

Here you have a full example:

```
define_function(sum)
{
  int64_t a = integer_argument(1);
  int64_t b = integer_argument(2);

  if (a == UNDEFINED || b == UNDEFINED)
    return_integer(UNDEFINED);

  return_integer(a + b);
}

begin_declarations;

    declare_function("sum", "ii", "i", sum);

end_declarations;
```

As you can see in the example above, your function code must be defined before the declaration section, like this:

```
define_function(<function identifier>)
{
  ...your code here
}
```

We are going to discuss function implementation more in depth in the *More about functions* section.

4.3 Implementing the module's logic

Every module must implement two functions which are called by YARA during the scanning of a file or process memory space: `module_load` and `module_unload`. Both functions are called once for each scanned file or process, but only if the module was imported by means of the `import` directive. If the module is not imported by some rule neither `module_load` nor `module_unload` will be called.

The `module_load` function has the following prototype:

```
int module_load(
    YR_SCAN_CONTEXT* context,
    YR_OBJECT* module,
    void* module_data,
    size_t module_data_size)
```

The `context` argument contains information relative to the current scan, including the data being scanned. The `module` argument is a pointer to a `YR_OBJECT` structure associated to the module. Each structure, variable or function declared in a YARA module is represented by a `YR_OBJECT` structure. These structures conform a tree whose root is the module's `YR_OBJECT` structure. If you have the following declarations in a module named *mymodule*:

```
begin_declarations;

    declare_integer("foo");

    begin_struct("bar");

        declare_string("baz");

    end_struct("bar");

end_declarations;
```

Then the tree will look like this:

```
YR_OBJECT(type=OBJECT_TYPE_STRUCT, name="mymodule")
 |
 |_ YR_OBJECT(type=OBJECT_TYPE_INTEGER, name="foo")
 |
 |_ YR_OBJECT(type=OBJECT_TYPE_STRUCT, name="bar")
     |
     |_ YR_OBJECT(type=OBJECT_TYPE_STRING, name="baz")
```

Notice that both *bar* and *mymodule* are of the same type `OBJECT_TYPE_STRUCT`, which means that the `YR_OBJECT` associated to the module is just another structure like *bar*. In fact, when you write in your rules something like `mymodule.foo` you're performing a field lookup in a structure in the same way that `bar.baz` does.

In resume, the `module` argument allows you to access every variable, structure or function declared by the module by providing a pointer to the root of the objects tree.

The `module_data` argument is a pointer to any additional data passed to the module, and `module_data_size` is the size of that data. Not all modules require additional data, most of them rely on the data being scanned alone, but a few of them require more information as input. The *Cuckoo module* is a good example of this, it receives a behavior report associated to PE files being scanned which is passed in the `module_data` and `module_data_size` arguments.

For more information on how to pass additional data to your module take a look at the `-x` argument in *Running YARA from the command-line*.

4.3.1 Accessing the scanned data

Most YARA modules needs to access the file or process memory being scanned to extract information from it. The data being scanned is sent to the module in the `YR_SCAN_CONTEXT` structure passed to the `module_load` function. The data is sometimes sliced in blocks, therefore your module needs to iterate over the blocks by using the `foreach_memory_block` macro:

```
int module_load(
    YR_SCAN_CONTEXT* context,
    YR_OBJECT* module,
    void* module_data,
    size_t module_data_size)
{
    YR_MEMORY_BLOCK* block;

    foreach_memory_block(context, block)
    {
        ..do something with the current memory block
    }
}
```

Each memory block is represented by a `YR_MEMORY_BLOCK` structure with the following attributes:

uint8_t* **data**
    Pointer to the actual data for this memory block.

size_t **size**
    Size of the data block.

size_t **base**
    Base offset/address for this block. If a file is being scanned this field contains the offset within the file where the block begins, if a process memory space is being scanned this contains the virtual address where the block begins.

The blocks are always iterated in the same order as they appear in the file or process memory. In the case of files the first block will contain the beginning of the file. Actually, a single block will contain the whole file's content in most cases, but you can't rely on that while writing your code. For very big files YARA could eventually split the file into two or more blocks, and your module should be prepared to handle that.

The story is very different for processes. While scanning a process memory space your module will definitely receive a large number of blocks, one for each committed memory region in the proccess address space.

However, there are some cases where you don't actually need to iterate over the blocks. If your module just parses the header of some file format you can safely assume that the whole header is contained within the first block (put some checks in your code nevertheless). In those cases you can use the `first_memory_block` macro:

```
int module_load(
    YR_SCAN_CONTEXT* context,
    YR_OBJECT* module,
    void* module_data,
    size_t module_data_size)
{
    YR_MEMORY_BLOCK* block;

    block = first_memory_block(context);

    ..do something with the memory block
}
```

4.3.2 Setting variable's values

The `module_load` function is where you assign values to the variables declared in the declarations section, once you've parsed or analized the scanned data and/or any additional module's data. This is done by using the `set_integer` and `set_string` functions:

void **set_integer** (int64_t *value*, YR_OBJECT* *object*, char* *field*, ...)

void **set_string** (char* *value*, YR_OBJECT* *object*, char* *field*, ...)

Both functions receive a value to be assigned to the variable, a pointer to a `YR_OBJECT` representing the variable itself or some ancestor of that variable, a field descriptor, and additional arguments as defined by the field descriptor.

If we are assigning the value to the variable represented by `object` itself, then the field descriptor must be `NULL`. For example, assuming that `object` points to a `YR_OBJECT` structure corresponding to some integer variable, we can set the value for that integer variable with:

```
set_integer(<value>, object, NULL);
```

The field descriptor is used when you want to assign the value to some descendant of `object`. For example, consider the following declarations:

```
begin_declarations;

    begin_struct("foo");

        declare_string("bar");

        begin_struct("baz");

            declare_integer("qux");

        end_struct("baz");

    end_struct("foo");

end_declarations;
```

If `object` points to the `YR_OBJECT` associated to the `foo` structure you can set the value for the `bar` string like this:

```
set_string(<value>, object, "bar");
```

And the value for `qux` like this:

```
set_integer(<value>, object, "baz.qux");
```

Do you remember that the `module` argument for `module_load` was a pointer to a `YR_OBJECT`? Do you remember that this `YR_OBJECT` is an structure just like `bar` is? Well, you could also set the values for `bar` and `qux` like this:

```
set_string(<value>, module, "foo.bar");
set_integer(<value>, module, "foo.baz.qux");
```

But what happens with arrays? How can I set the value for array items? If you have the following declarations:

```
begin_declarations;

    declare_integer_array("foo");

    begin_struct_array("bar")

        declare_string("baz");
        declare_integer_array("qux");

    end_struct_array("bar");

end_declarations;
```

Then the following statements are all valid:

```
set_integer(<value>, module, "foo[0]");
set_integer(<value>, module, "foo[%i]", 2);
set_string(<value>, module, "bar[%i].baz", 5);
set_string(<value>, module, "bar[0].qux[0]");
set_string(<value>, module, "bar[0].qux[%i]", 0);
set_string(<value>, module, "bar[%i].qux[%i]", 100, 200);
```

Those `%i` in the field descriptor are replaced by the additional integer arguments passed to the function. This work in the same way than `printf` in C programs, but the only format specifier accepted is `%i`.

If you don't explicitely assign a value to a declared variable or array item it will remain in undefined state. That's not a problem at all, and is even useful in many cases. For example, if your module parses files from certain format and it receives one from a different format, you can safely leave all your variables undefined instead of assigning them bogus values that doesn't make sense. YARA will handle undefined values in rule conditions as described in *Using modules*.

In addition to `set_integer` and `set_string` functions you have their `get_integer` and `get_string` counterparts. As the names suggest they are used for getting the value of a variable, which can be useful in the implementation of your functions to retrieve values previously stored by `module_load`.

int64_t **`get_integer`**(YR_OBJECT* *object*, char* *field*, ...)

char* **`get_string`**(YR_OBJECT* *object*, char* *field*, ...)

There's also a function to the get any `YR_OBJECT` in the objects tree:

YR_OBJECT* **`get_object`**(YR_OBJECT* *object*, char* *field*, ...)

Here goes a little exam...

Are the following two lines equivalent? Why?

```
set_integer(1, get_object(module, "foo.bar"), NULL);
set_integer(1, module, "foo.bar");
```

4.3.3 Storing data for later use

Sometimes the information stored directly in your variables by means of `set_integer` and `set_string` is not enough. You may need to store more complex data structures or information that don't need to be exposed to YARA rules.

Storing information is essential when your module exports functions to be used in YARA rules. The implementation of these functions usually require to access information generated by `module_load` which must kept somewhere. You may be tempted to define global variables where to put the required information, but this would make your code non-thread-safe. The correct approach is using the `data` field of the `YR_OBJECT` structures.

Each `YR_OBJECT` has a `void* data` field which can be safely used by your code to store a pointer to any data you may need. A typical pattern is using the `data` field of the module's `YR_OBJECT`, like in the following example:

```
typedef struct _MY_DATA
{
   int some_integer;

} MY_DATA;

int module_load(
    YR_SCAN_CONTEXT* context,
    YR_OBJECT* module,
    void* module_data,
    size_t module_data_size)
{
    module->data = yr_malloc(sizeof(MY_DATA));
    ((MY_DATA*) module->data)->some_integer = 0;

    return ERROR_SUCCESS;
}
```

Don't forget to release the allocated memory in the `module_unload` function:

```
int module_unload(
    YR_OBJECT* module)
{
    yr_free(module->data);

    return ERROR_SUCCESS;
}
```

Warning: Don't use global variables for storing data. Functions in a module can be invoked from different threads at the same time and data corruption or misbehavior can occur.

4.4 More about functions

We already showed how to declare a function in *The declaration section*. Here we are going to discuss how to provide an implementation for them.

4.4.1 Function arguments

Within the function's code you get its arguments by using `integer_argument(n)`, `string_argument(n)` or `regexp_argument(n)` depending on the type of the argument, where *n* is the 1-based argument's number.

If your function receives a string, a regular expression and an integer in that order, you can get their values with:

```
char* arg_1 = string_argument(1);
RE_CODE arg_2 = regexp_argument(2);
int64_t arg_3 = integer_argument(3);
```

Notice that the C type for integer arguments is `int64_t` and for regular expressions is `RE_CODE`.

4.4.2 Return values

Functions can return two types of values: strings and integers. Instead of using the C *return* statement you must use `return_string(x)` or `return_integer(x)` to return from a function, depending on the function's return type. In both cases *x* is a constant, variable, or expression evaluating to `char*` or `int64_t` respectively.

You can use `return_string(UNDEFINED)` and `return_integer(UNDEFINED)` to return undefined values from the function. This is useful in many situations, for example if the arguments passed to the functions don't make sense, or if your module expects a particular file format and the scanned file is from another format, or in any other case where your function can't a return a valid value.

Warning: Don't use the C *return* statement for returning from a function. The returned value will be interpreted as an error code.

4.4.3 Accessing objects

While writing a function we sometimes need to access values previously assigned to module's variables, or additional data stored in the `data` field of `YR_OBJECT` structures as discussed earlier in *Storing data for later use*. But for that we need a way to get access to the corresponding `YR_OBJECT` first. There are two functions to do that: `module()` and `parent()`. The `module()` function returns a pointer to the top-level `YR_OBJECT` corresponding to the module, the same one passed to the `module_load` function. The `parent()` function returns a pointer to the `YR_OBJECT` corresponding to the structure where the function is contained. For example, consider the following code snipet:

```
define_function(f1)
{
    YR_OBJECT* module = module();
    YR_OBJECT* parent = parent();

    // parent == module;
}

define_function(f2)
{
    YR_OBJECT* module = module();
    YR_OBJECT* parent = parent();

    // parent != module;
}

begin_declarations;

    declare_function("f1", "i", "i", f1);

    begin_struct("foo");

        declare_function("f2", "i", "i", f2);

    end_struct("foo");

end_declarations;
```

In `f1` the `module` variable points to the top-level `YR_OBJECT` as well as the `parent` variable, because the parent for `f1` is the module itself. In `f2` however the `parent` variable points to the `YR_OBJECT` corresponding to the `foo` structure while `module` points to the top-level `YR_OBJECT` as before.

4.4.4 Scan context

From within a function you can also access the `YR_SCAN_CONTEXT` structure discussed earlier in *Accessing the scanned data*. This is useful for functions which needs to inspect the file or process memory being scanned. This is how you get a pointer to the `YR_SCAN_CONTEXT` structure:

```
YR_SCAN_CONTEXT* context = scan_context();
```

CHAPTER 5

Running YARA from the command-line

In order to invoke YARA you'll need two things: a file with the rules you want to use (either in source code or compiled form) and the target to be scanned. The target can be a file, a folder, or a process.

```
yara [OPTIONS] RULES_FILE TARGET
```

Rule files can be passed directly in source code form, or can be previously compiled with the `yarac` tool. You may prefer to use your rules in compiled form if you are going to invoke YARA multiple times with the same rules. This way you'll save time, because for YARA is faster to load compiled rules than compiling the same rules over and over again.

The rules will be applied to the target specified as the last argument to YARA, if it's a path 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 for that.

Available options are:

-t `<tag>`
    Print rules tagged as <tag> and ignore the rest.

-i `<identifier>`
    Print rules named <identifier> and ignore the rest.

-n
    Print not satisfied rules only (negate).

-g
    Print tags.

-m
    Print metadata.

-s
    Print matching strings.

-p `<number>`
    Use the specified <number> of threads to scan a directory.

-l `<number>`
    Abort scanning after matching a number of rules.

-a `<seconds>`
    Abort scanning after a number of seconds has elapsed.

-d `<identifier>=<value>`
    Define external variable.

-x `<module>=<file>`
: Pass file's content as extra data to module.

-r
: Recursively search for directories.

-f
: Fast matching mode.

-w
: Disable warnings.

-v
: Show version information.

Here you have some examples:

- Apply rules on */foo/bar/rules1* and */foo/bar/rules2* to all files on current directory. Subdirectories are not scanned:

  ```
  yara /foo/bar/rules1 /foo/bar/rules2 .
  ```

- Apply rules on */foo/bar/rules* to *bazfile*. Only reports rules tagged as *Packer* or *Compiler*:

  ```
  yara -t Packer -t Compiler /foo/bar/rules bazfile
  ```

- Scan all files in the */foo* directory and its subdirectories:

  ```
  yara -r /foo
  ```

- Defines three external variables *mybool*, *myint* and *mystring*:

  ```
  yara -d mybool=true -d myint=5 -d mystring="my string" /foo/bar/rules bazfile
  ```

- Apply rules on */foo/bar/rules* to *bazfile* while passing the content of *cuckoo_json_report* to the cuckoo module:

  ```
  yara -x cuckoo=cuckoo_json_report /foo/bar/rules bazfile
  ```

CHAPTER 6

Using YARA from Python

YARA can be also used from Python through the `yara-python` library. Once the library is built and installed as described in *Compiling and installing YARA* you'll have access to the full potential of YARA from your Python scripts.

The first step is importing the YARA library:

```
import yara
```

Then you will need to compile your YARA rules before applying them to your data, the rules can be compiled from a file path:

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

The default argument is filepath, so you don't need to explicitly specify its name:

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

You can also compile your rules from a file object:

```
fh = open('/foo/bar/myrules')
rules = yara.compile(file=fh)
fh.close()
```

Or you can compile them directly from a Python string:

```
rules = yara.compile(source='rule dummy { condition: true }')
```

If you want to compile a group of files or strings at the same time you can do it by using the filepaths or sources named arguments:

```
rules = yara.compile(filepaths={

  'namespace1':'/my/path/rules1',
  'namespace2':'/my/path/rules2'
})

rules = yara.compile(sources={

  'namespace1':'rule dummy { condition: true }',
  'namespace2':'rule dummy { condition: false }'
})
```

Notice that both `filepaths` and `sources` must be dictionaries with keys of string type. The dictionary keys are used as a namespace identifier, allowing to differentiate between rules with the same name in different sources, as occurs in the second example with the *dummy* name.

The `compile` method also have an optional boolean parameter named `includes` which allows you to control whether or not the include directive should be accepted in the source files, for example:

```
rules = yara.compile('/foo/bar/my_rules', includes=False)
```

If the source file contains include directives the previous line would raise an exception.

If you are using external variables in your rules you must define those externals variables either while compiling the rules, or while applying the rules to some file. To define your variables at the moment of compilation you should pass the `externals` parameter to the `compile` method. For example:

```
rules = yara.compile('/foo/bar/my_rules',
  externals= {'var1': 'some string', 'var2': 4, 'var3': True})
```

The `externals` parameter must be a dictionary with the names of the variables as keys and an associated value of either string, integer or boolean type.

The `compile` method also accepts the optional boolean argument `error_on_warning`. This arguments tells YARA to raise an exception when a warning is issued during compilation. Such warnings are typically issued when your rules contains some construct that could be slowing down the scanning. The default value for the `error_on_warning` argument is False.

In all cases `compile` returns an instance of the class `yara.Rules` Rules. This class have a `save` method that can be used to save the compiled rules to a file:

```
rules.save('/foo/bar/my_compiled_rules')
```

The compiled rules can be loaded later by using the `load` method:

```
rules = yara.load('/foo/bar/my_compiled_rules')
```

The result of `load` is also an instance of the class `yara.Rules`.

Instances of `Rules` also have a `match` method, which allows to apply the rules to a file:

```
matches = rules.match('/foo/bar/my_file')
```

But you can also apply the rules to a Python string:

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

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

Or to a running process:

```
matches = rules.match(pid=1234)
```

As in the case of `compile`, the `match` method can receive definitions for externals variables in the `externals` argument.

```
matches = rules.match('/foo/bar/my_file',
  externals= {'var1': 'some other string', 'var2': 100})
```

Externals variables defined during compile-time don't need to be defined again in subsequent calls to the `match` method. However you can redefine any variable as needed, or provide additional definitions that weren't provided during compilation.

In some situations involving a very large set of rules or huge files the `match` method can take too much time to run. In those situations you may find useful the `timeout` argument:

```
matches = rules.match('/foo/bar/my_huge_file', timeout=60)
```

If the `match` function does not finish before the specified number of seconds elapsed, a `TimeoutError` exception is raised.

You can also specify a callback function when invoking `match` method. The provided function will be called for every rule, no matter if matching or not. Your callback function should expect a single parameter of dictionary type, and should return `CALLBACK_CONTINUE` to proceed to the next rule or `CALLBACK_ABORT` to stop applying rules to your data.

Here is an example:

```
import yara

def mycallback(data):
  print data
  yara.CALLBACK_CONTINUE

matches = rules.match('/foo/bar/my_file', callback=mycallback)
```

The passed dictionary will be something like this:

```
{
  'tags': ['foo', 'bar'],
  'matches': True,
  'namespace': 'default',
  'rule': 'my_rule',
  'meta': {},
  'strings': [(81L, '$a', 'abc'), (141L, '$b', 'def')]
}
```

The *matches* field indicates if the rules matches the data or not. The *strings* fields is a list of matching strings, with vectors of the form:

```
(<offset>, <string identifier>, <string data>)
```

The `match` method returns a list of instances of the class `Match`. Instances of this class have the same attributes as the dictionary passed to the callback function.

6.1 Reference

`yara.`**`compile`**`(...)`

Compile YARA sources.

One of *filepath*, *source*, *file*, *filepaths* or *sources* must be provided. The remaining arguments are optional.

Parameters

- **filepath** (*str*) – Path to the source file.
- **source** (*str*) – String containing the rules code.
- **file** (*file*) – Source file as a file object.
- **filepaths** (*dict*) – Dictionary where keys are namespaces and values are paths to source files.
- **sources** (*dict*) – Dictionary where keys are namespaces and values are strings containing rules code.
- **externals** (*dict*) – Dictionary with external variables. Keys are variable names and values are variable values.

- **includes** (*boolean*) – True if include directives are allowed or False otherwise. Default value: *True*.
- **error_on_warning** (*boolean*) – If true warnings are treated as errors, raising an exception.

Returns Compiled rules object.

Return type `yara.Rules`

Raises

- **YaraSyntaxError** – If a syntax error was found.
- **YaraError** – If an error occurred.

`yara.`**`load`**(*filepath*)

Load compiled rules from a file.

Parameters **filepath** (*str*) – Path to the file.

Returns Compiled rules object.

Return type `yara.Rules`

Raises **YaraError**: If an error occurred while loading the file.

class `yara.`**`Rules`**

Instances of this class are returned by `yara.compile()` and represents a set of compiled rules.

`match`(*filepath*, *pid*, *data*, *externals=None*, *callback=None*, *fast=False*, *timeout=None*, *modules_data=None*)

Scan a file, process memory or data string.

One of *filepath*, *pid* or *data* must be provided. The remaining arguments are optional.

Parameters

- **filepath** (*str*) – Path to the file to be scanned.
- **pid** (*int*) – Process id to be scanned.
- **data** (*str*) – Data to be scanned.
- **externals** (*dict*) – Dictionary with external variables. Keys are variable names and values are variable values.
- **callback** (*function*) – Callback function invoked for each rule.
- **fast** (*bool*) – If true performs a fast mode scan.
- **timeout** (*int*) – Aborts the scanning when the number of specified seconds have elapsed.
- **modules_data** (*dict*) – Dictionary with additional data to modules. Keys are module names and values are *bytes* objects containing the additional data.

Raises

- **YaraTimeoutError** – If the timeout was reached.
- **YaraError** – If an error occurred during the scan.

`save`(*filepath*)

Save compiled rules to a file.

Parameters **filepath** (*str*) – Path to the file.

Raises **YaraError**: If an error occurred while saving the file.

CHAPTER 7

The C API

You can integrate YARA into your C/C++ project by using the API privided by the *libyara* library. This API gives you access to every YARA feature and it's the same API used by the command-line tools `yara` and `yarac`.

7.1 Initalizing and finalizing *libyara*

The first thing your program must do when using *libyara* is initializing the library. This is done by calling the `yr_initialize()` function. This function allocates any resources needed by the library and initalizes internal data structures. Its counterpart is `yr_finalize()`, which must be called when you are finished using the library.

In a multi-threaded program only the main thread must call `yr_initialize()` and `yr_finalize()`, but any additional thread using the library must call `yr_finalize_thread()` before exiting.

7.2 Compiling rules

Before using your rules to scan any data you need to compile them into binary form. For that purpose you'll need a YARA compiler, which can be created with `yr_compiler_create()`. After being used, the compiler must be destroyed with `yr_compiler_destroy()`.

You can use either `yr_compiler_add_file()` or `yr_compiler_add_string()` to add one or more input sources to be compiled. Both of these functions receive an optional namespace. Rules added under the same namespace behaves as if they were contained within the same source file or string, so, rule identifiers must be unique among all the sources sharing a namespace. If the namespace argument is `NULL` the rules are put in the *default* namespace.

Both `yr_compiler_add_file()` and `yr_compiler_add_string()` return the number of errors found in the source code. If the rules are correct they will return 0. For more detailed error information you must set a callback function by using `yr_compiler_set_callback()` before calling `yr_compiler_add_file()` or `yr_compiler_add_string()`. The callback function has the following prototype:

```
void callback_function(
    int error_level,
    const char* file_name,
    int line_number,
    const char* message)
```

Possible values for `error_level` are `YARA_ERROR_LEVEL_ERROR` and `YARA_ERROR_LEVEL_WARNING`. The arguments `file_name` and `line_number` contains the file name and line number where the error or warning occurs. `file_name` is the one passed to `yr_compiler_add_file()`. It can be `NULL` if you passed `NULL` or if you're using `yr_compiler_add_string()`.

After you successfully added some sources you can get the compiled rules using the `yr_compiler_get_rules()` function. You'll get a pointer to a `YR_RULES` structure which can be used to scan your data as described in *Scanning data*. Once `yr_compiler_get_rules()` is invoked you can not add more sources to the compiler, but you can get multiple instances of the compiled rules by calling `yr_compiler_get_rules()` multiple times.

Each instance of `YR_RULES` must be destroyed with `yr_rules_destroy()`.

7.3 Saving and retrieving compiled rules

Compiled rules can be saved to a file and retrieved later by using `yr_rules_save()` and `yr_rules_load()`. Rules compiled and saved in one machine can be loaded in another machine as long as they have the same endianness, no matter the operating system or if they are 32-bits or 64-bits systems. However files saved with older versions of YARA may not work with newer version due to changes in the file layout.

7.4 Scanning data

Once you have an instance of `YR_RULES` you can use it to scan data either from a file or a memory buffer with `yr_rules_scan_file()` and `yr_rules_scan_mem()` respectively. The results from the scan are notified to your program via a callback function. The callback has the following prototype:

```
int callback_function(
    int message,
    void* message_data,
    void* user_data);
```

Possible values for `message` are:

```
CALLBACK_MSG_RULE_MATCHING
CALLBACK_MSG_RULE_NOT_MATCHING
CALLBACK_MSG_SCAN_FINISHED
CALLBACK_MSG_IMPORT_MODULE
```

Your callback function will be called once for each existing rule with either a `CALLBACK_MSG_RULE_MATCHING` or `CALLBACK_MSG_RULE_NOT_MATCHING` message, depending if the rule is matching or not. In both cases a pointer to the `YR_RULE` structure associated to the rule is passed in the `message_data` argument. You just need to perform a typecast from `void*` to `YR_RULE*` to access the structure.

The callback is also called once for each imported module, with the `CALLBACK_MSG_IMPORT_MODULE` message. In this case `message_data` points to a `YR_MODULE_IMPORT` structure. This structure contains a `module_name` field pointing to a null terminated string with the name of the module being imported and two other fields `module_data` and `module_data_size`. These fields are initially set to `NULL` and `0` , but your program can assign a pointer to some arbitrary data to `module_data` while setting `module_data_size` to the size of the data. This way you can pass additional data to those modules requiring it, like the *Cuckoo module* for example.

Lastly, the callback function is also called with the `CALLBACK_MSG_SCAN_FINISHED` message when the scan is finished. In this case `message_data` is `NULL`.

In all cases the `user_data` argument is the same passed to `yr_rules_scan_file()` or `yr_rules_scan_mem()`. This pointer is not touched by YARA, it's just a way for your program to pass arbitrary data to the callback function.

Both `yr_rules_scan_file()` and `yr_rules_scan_mem()` receive a `flags` argument and a `timeout` argument. The only flag defined at this time is `SCAN_FLAGS_FAST_MODE`, so you must pass either this flag or a zero value. The `timeout` argument forces the function to return after the specified number of seconds aproximately, with a zero meaning no timeout at all.

The `SCAN_FLAGS_FAST_MODE` flag makes the scanning a little faster by avoiding multiple matches of the same string when not necessary. Once the string was found in the file it's subsequently ignored, implying that you'll have a single match for the string, even if it appears multiple times in the scanned data. This flag has the same effect of the `-f` command-line option described in *Running YARA from the command-line*.

7.5 API reference

7.5.1 Data structures

`YR_COMPILER`
Data structure representing a YARA compiler.

`YR_RULES`
Data structure representing a set of compiled rules.

`YR_RULE`
Data structure representing a single rule.

const char* **`identifier`**
Rule identifier.

const char* **`tags`**
Pointer to a sequence of null terminated strings with tag names. An additional null character marks the end of the sequence. Example: `tag1\0tag2\0tag3\0\0`. To iterate over the tags you can use `yr_rule_tags_foreach()`.

YR_META* **`metas`**
Pointer to a sequence of `YR_META` structures. To iterate over the structures use `yr_rule_metas_foreach()`.

YR_STRING* **`strings`**
Pointer to a sequence of `YR_STRING` structures. To iterate over the structures use `yr_rule_strings_foreach()`.

`YR_META`
Data structure representing a metadata value.

const char* **`identifier`**
Meta identifier.

int32_t **`type`**
One of the following metadata types:

`META_TYPE_NULL` `META_TYPE_INTEGER` `META_TYPE_STRING` `META_TYPE_BOOLEAN`

`YR_STRING`
Data structure representing a string declared in a rule.

const char* **`identifier`**
String identifier.

`YR_MATCH`
Data structure representing a string match.

int64_t **`base`**
Base offset/address for the match. While scanning a file this field is usually zero, while scanning a process memory space this field is the virtual address of the memory block where the match was found.

int64_t **offset**
Offset of the match relative to *base*.

int32_t **length**
Length of the matching string

uint8_t* **data**
Pointer to the matching string.

YR_MODULE_IMPORT

const char* **module_name**
Name of the module being imported.

void* **module_data**
Pointer to additional data passed to the module. Initially set to `NULL`, your program is responsible of setting this pointer while handling the CALLBACK_MSG_IMPORT_MODULE message.

size_t **module_data_size**
Size of additional data passed to module. Your program must set the appropriate value if `module_data` is modified.

7.5.2 Functions

void **yr_initialize** (void)
Initalize the library. Must be called by the main thread before using any other function.

void **yr_finalize** (void)
Finalize the library. Must be called by the main free to release any resource allocated by the library.

void **yr_finalize_thread** (void)
Any thread using the library, except the main thread, must call this function when it finishes using the library.

int **yr_compiler_create** (YR_COMPILER** *compiler*)
Create a YARA compiler. You must pass the address of a pointer to a `YR_COMPILER`, the function will set the pointer to the newly allocated compiler. Returns one of the following error codes:

`ERROR_SUCCESS`

`ERROR_INSUFICENT_MEMORY`

void **yr_compiler_destroy** (YR_COMPILER* *compiler*)
Destroy a YARA compiler.

void **yr_compiler_set_callback** (YR_COMPILER* *compiler*, YR_COMPILER_CALLBACK_FUNC *callback*)
Set a callback for receiving error and warning information.

int **yr_compiler_add_file** (YR_COMPILER* *compiler*, FILE* *file*, const char* *namespace*, const char* *file_name*)
Compile rules from a *file*. Rules are put into the specified *namespace*, if *namespace* is `NULL` they will be put into the default namespace. *file_name* is the name of the file for error reporting purposes and can be set to `NULL`. Returns the number of errors found during compilation.

int **yr_compiler_add_string** (YR_COMPILER* *compiler*, const char* *string*, const char* *namespace_*)
Compile rules from a *string*. Rules are put into the specified *namespace*, if *namespace* is `NULL` they will be put into the default namespace. Returns the number of errors found during compilation.

int **yr_compiler_get_rules** (YR_COMPILER* *compiler*, YR_RULES** *rules*)
Get the compiled rules from the compiler. Returns one of the following error codes:

ERROR_SUCCESS

ERROR_INSUFICENT_MEMORY

void **yr_rules_destroy**(YR_RULES* *rules*)

Destroy compiled rules.

int **yr_rules_save**(YR_RULES* *rules*, const char* *filename*)

Save *rules* into the file specified by *filename*. Returns one of the following error codes:

ERROR_SUCCESS

ERROR_COULD_NOT_OPEN_FILE

int **yr_rules_load**(const char* *filename*, YR_RULES** *rules*)

Load rules from the file specified by *filename*. Returns one of the following error codes:

ERROR_SUCCESS

ERROR_INSUFICENT_MEMORY

ERROR_COULD_NOT_OPEN_FILE

ERROR_INVALID_FILE

ERROR_CORRUPT_FILE

ERROR_UNSUPPORTED_FILE_VERSION

ERROR_INSUFICENT_MEMORY

int **yr_rules_scan_mem**(YR_RULES* *rules*, uint8_t* *buffer*, size_t *buffer_size*, int *flags*, YR_CALLBACK_FUNC *callback*, void* *user_data*, int *timeout*)

Scan a memory buffer. Returns one of the following error codes:

ERROR_SUCCESS

ERROR_INSUFICENT_MEMORY

ERROR_TOO_MANY_SCAN_THREADS

ERROR_SCAN_TIMEOUT

ERROR_CALLBACK_ERROR

ERROR_TOO_MANY_MATCHES

int **yr_rules_scan_file**(YR_RULES* *rules*, const char* *filename*, int *flags*, YR_CALLBACK_FUNC *callback*, void* *user_data*, int *timeout*)

Scan a file. Returns one of the following error codes:

ERROR_SUCCESS

ERROR_INSUFICENT_MEMORY

ERROR_COULD_NOT_MAP_FILE

ERROR_ZERO_LENGTH_FILE

ERROR_TOO_MANY_SCAN_THREADS

ERROR_SCAN_TIMEOUT

ERROR_CALLBACK_ERROR

ERROR_TOO_MANY_MATCHES

yr_rule_tags_foreach (rule, tag)

Iterate over the tags of a given rule running the block of code that follows each time with a different value for *tag* of type `const char*`. Example:

```
const char* tag;

/* rule is a YR_RULE object */

yr_rule_tags_foreach(rule, tag)
{
  ..do something with tag
}
```

yr_rule_metas_foreach (rule, meta)

Iterate over the `YR_META` structures associated to a given rule running the block of code that follows each time with a different value for *meta*. Example:

```
YR_META* meta;

/* rule is a YR_RULE object */

yr_rule_metas_foreach(rule, meta)
{
  ..do something with meta
}
```

yr_rule_strings_foreach (rule, string)

Iterate over the `YR_STRING` structures associated to a given rule running the block of code that follows each time with a different value for *string*. Example:

```
YR_STRING* string;

/* rule is a YR_RULE object */

yr_rule_strings_foreach(rule, string)
{
  ..do something with string
}
```

yr_string_matches_foreach (string, match)

Example:

```
YR_MATCH* match;

/* string is a YR_STRING object */

yr_string_matches_foreach(string, match)
{
  ..do something with match
}
```

7.5.3 Error codes

ERROR_SUCCESS

Everything went fine.

ERROR_INSUFICENT_MEMORY

Insuficient memory to complete the operation.

ERROR_COULD_NOT_OPEN_FILE
File could not be opened.

ERROR_COULD_NOT_MAP_FILE
File could not be mapped into memory.

ERROR_ZERO_LENGTH_FILE
File length is zero.

ERROR_INVALID_FILE
File is not a valid rules file.

ERROR_CORRUPT_FILE
Rules file is corrupt.

ERROR_UNSUPPORTED_FILE_VERSION
File was generated by a different YARA and can't be loaded by this version.

ERROR_TOO_MANY_SCAN_THREADS
Too many threads trying to use the same `YR_RULES` object simultaneosly. The limit is defined by `MAX_THREADS` in *./include/yara/limits.h*

ERROR_SCAN_TIMEOUT
Scan timed out.

ERROR_CALLBACK_ERROR
Callback returned an error.

ERROR_TOO_MANY_MATCHES
Too many matches for some string in your rules. This usually happens when your rules contains very short or very common strings like `01 02` or `FF FF FF FF`. The limit is defined by `MAX_STRING_MATCHES` in *./include/yara/limits.h*

Python Module Index