yara-ctypes Documentation

Release 1.7.7

Michael Dorman

Mar 06, 2017

Contents

1 Getting started **3**
- 1.1 Install guide . . . 3
- 1.2 How to scan using yara-ctypes `yara.scan` . . . 4
- 1.3 Building libyara-1.6 for yara-ctypes . . . 7

2 Reference **13**
- 2.1 `yara.scan` — Thread pool execution of rules matching . . . 13
- 2.2 `yara.cli` — A command line YARA rules scanning utility . . . 14
- 2.3 `yara.rules` — YARA namespaces, compilation, and matching . . . 14
- 2.4 `yara.libyara_wrapper` — ctypes wrapper for libyara . . . 17

3 Indices and tables **19**

Python Module Index **21**

What is yara-ctypes:

- A powerful python wrapper for yara-project's libyara v1.6.
- Supports thread safe matching of YARA rules.
- namespace management to allow easy loading of multiple YARA rules into a single libyara context.
- Comes with a scan module which exposes a user CLI and demonstrates a pattern for executing match jobs across a thread pool.

Why:

- ctypes releases the GIL on system function calls... Run your PC to its true potential.
- No more building the PyC extension...
- I found a few bugs and memory leaks and wanted to make my life simple.

As a reference and guide to yara-ctypes see: yara-ctypes documentation

For additional tips / tricks with this wrapper feel free to post a question at the github yara-ctypes/issues page.

Project hosting provided by github.com.

[mjdorma+yara-ctypes@gmail.com]

CHAPTER 1

Getting started

Install guide

Things to know about installing yara-ctypes.

PyPi install

Simply run the following:

```
pip install yara
```

If you do not have pip, you can click here to find the latest download package.

Unzip than install:

```
python setup.py install
```

Download and install the master

You can find the master copy of yara-ctypes on github.

Here is how to install from the master:

```
wget -O master.zip https://github.com/mjdorma/yara-ctypes/zipball/master
unzip master.zip
cd mjdorma-yara-ctypes-XXX
python setup.py install
```

Missing a dll? Try installing MS VC++ 2010 redistributable package

The shipped dlls' were built using Visual Studio 2010. If you do not have the appropriate runtime already installed you will get an error message pop up saying you are missing `msvcr100.dll`. Download and install the appropriate redistribution package for your platform:

- Microsoft Visual C++ 2010 Redistributable Package (x86) (or vcredist_x86.exe)
- Microsoft Visual C++ 2010 Redistributable Package (x64) (or vcredist_x64.exe)

Failing to import libyara

At this point you need to figure out if the shipped library file is compatible with your system/platform. You may need to build your own libyara library from scratch.

See *Building libyara-1.6 for yara-ctypes* for more information.

How to scan using yara-ctypes `yara.scan`

This page should contain all of the information required to successfully operate *`yara.scan`* as a system scanning utility.

Executing `yara.cli`

Once yara-ctypes is installed into your Python environment you can run the scan module by executing the scan module as follows:

```
$ python -m yara.cli -h
```

or:

```
$ yara-ctypes -h
```

Performing a scan

List available modules:

```
$ yara-ctypes --list

Rules + example.packer_rules
      + hbgary.sockets
      + hbgary.libs
      + hbgary.compression
      + hbgary.fingerprint
      + hbgary.integerparsing
      + hbgary.antidebug
      + hbgary.microsoft
```

Scan process memory:

```
$ ps
  PID TTY          TIME CMD
 6975 pts/7    00:00:05 bash
13479 pts/7    00:00:00 ps

$ sudo yara-ctypes --proc 6975 > result.out

Rules + hbgary.compiler
      + example.packer_rules
      + hbgary.sockets
      + hbgary.libs
      + hbgary.compression
      + hbgary.fingerprint
      + hbgary.integerparsing
      + hbgary.antidebug
      + hbgary.microsoft
scan queue: 0       result queue: 0
scanned 1 items... done.

$ ls -lah result.out

-rw-rw-r-- 1 mick mick 222K Sep  1 17:36 result.out
```

Scan a file:

```
$ sudo yara-ctypes /usr/bin/ > result.out

Rules + hbgary.compiler
      + example.packer_rules
      + hbgary.sockets
      + hbgary.libs
      + hbgary.compression
      + hbgary.fingerprint
      + hbgary.integerparsing
      + hbgary.antidebug
      + hbgary.microsoft
scan queue: 0       result queue: 0
scanned 1518 items... done.

> ls -lah result.out

-rw-rw-r-- 1 mick mick 17M Sep  1 17:37 result.out
```

YARA rules files and folder

If you are not familiar with YARA rules files visit yara project to learn more.

To make life simple the *`yara.rules`* module supports filtered namespaced loading of multiple YARA rules files into a single context. This is managed through a translation of folder names and file names into '.' seperated names. The root of this folder structured is defined by the YARA_RULES path.

By default the YARA_RULES path points to the following path:

```
os.path.dirname(:mod:`yara.rules`.__file__) + '/rules'
```

Included rules folder

The rules folder shipped with yara-ctypes helps with testing and works as a good example set of YARA rules for people to get started from.

Packaged rules folder:

```
./rules/hbgary/libs.yar
./rules/hbgary/compression.yar
./rules/hbgary/fingerprint.yar
./rules/hbgary/microsoft.yar
./rules/hbgary/sockets.yar
./rules/hbgary/integerparsing.yar
./rules/hbgary/compiler.yar
./rules/hbgary/antidebug.yar
./rules/example/packer_rules.yar
```

Building a Rules object using `yara.load_rules()` will load all of the above yar files into the following namespaces:

```
hbgary.libs
hbgary.compression
hbgary.fingerprint
hbgary.microsoft
hbgary.sockets
hbgary.integerparsing
hbgary.compiler
hbgary.antidebug
example.packer_rules
```

Using yara-ctypes rules folders

This section will walk you through defining and loading a realistic rules folder.

A practical rules folder example:

We set out by defining two sub directories, one for our process memory specific signatures and the other for our file signatures.

Here is what it looks like:

```
~/rules/
    pid/loggers.yar
    pid/spammers.yar
    pid/infectors.yar
    file/loggers.yar
    file/spammers.yar
    file/infectors.yar
```

Accessing a rules folder:

To access our new rules folder we need to let *`yara.scan`* know where to look. We can do this by setting the env variable `YARA_RULES` to `export YARA_RULES=~/rules/`. Alternatively, we can specify the root of the rules folder with the input argument `--root=~/rules/`.

Confirm the rules are being loaded by *`yara.scan`*:

```
$ yara-ctypes --list
Rules + file.loggers
      + file.infectors
      + file.spammers
      + pid.spammers
      + pid.loggers
      + pid.infectors
```

Blacklisting and whitelisting namespaces:

Let's say we want to scan a bunch of files against all of the yar files under `~/rules/file/`. We can do this two ways. By either setting our `--whitelist=file` or setting our `--blacklist=pid`.

i.e.:

```
$ yara-ctypes --blacklist=pid --list
Rules + file.infectors
      + file.loggers
      + file.spammers
```

Whitelist and blacklist parameters are globbed out (*i.e. pid**).

The results are in and we find that `file.spammers` namespace is producing far too much noise. Let's remove `file.spammers` from scan too:

```
$ yara-ctypes --blacklist=pid,file.spamm --list
Rules + file.infectors
      + file.loggers
```

To demonstrate the namespace convetion further, we may find ourselves wanting to run a scan which includes `` `pid.spammers` ``. To do this we can simply run:

```
$ yara-ctypes --blacklist=file.spamm --whitelist=pid.spam,file --list
Rules + file.infectors
      + file.loggers
      + pid.spammers
```

Building libyara-1.6 for yara-ctypes

This guide captures some of the steps taken to make a clean checkout of tags/yara-1.6/ build and work for yara-ctypes.

Patch a clean checkout of yara-1.6

Checkout yara-1.6.0 from:

```
svn co http://yara-project.googlecode.com/svn/tags/yara-1.6.0 .
```

Modify the following two files from `./libyara/` to allow *`yara.rules`* cleanup after each search:

```
>>>yara.h<<<
+ void yr_free_matches(YARA_CONTEXT* context);

>>>libyara.c<<<
+ void yr_free_matches(YARA_CONTEXT* context)
+ {
```

```
+    RULE* rule;
+    STRING* string;
+    MATCH* match;
+    MATCH* next_match;
+    rule = context->rule_list.head;
+    while (rule != NULL)
+    {
+        string = rule->string_list_head;
+
+        while (string != NULL)
+        {
+            match = string->matches_head;
+            while (match != NULL)
+            {
+                next_match = match->next;
+                yr_free(match->data);
+                yr_free(match);
+                match = next_match;
+            }
+            string->matches_head = NULL;
+            string->matches_tail = NULL;
+            string = string->next;
+        }
+        rule = rule->next;
+    }
+ }
```

Building for Ubuntu

Install the development pre-requisites:

```
> sudo apt-get install build-essential flex libpcre3-dev libpcre3 bison
```

First attempt:

```
> cd $ROOTDIR/yara-1.6/
> ./configure
> make
```

If that fails, try to reconfigure:: > aclocal > automake -ac > autoheader > autoconf > ./configure make

Thats it, nice and easy...

Building for Windows

Build using Mingw32

Install prerequisites:

```
> install mingw32
> pcre-8.20 builds fine...  ./configure && make install
```

Run the build:

```
> autoreconf -fiv # force an autoreconf (or update/replace libtools m4)
> install build auto tools (including autoconf autogen)
> find the latest pcre and bison - build them! :P
> cd $ROOTDIR/yara-1.6/
> ./configure
> make
```

This will get you a 32bit dll. If you figure out how to do it under mingw64, let me know...

Build under Visual Studios

To build using Visual Studio, the following settings were added to the `windows/libyara/libyara.vcproj` Properties Page.

- [General][Configuration Type] = "Dynamic Library (.dll)"
- [C/C++][Runtime Library] = "Multi-threaded DLL (/MD)"

The *C/C++* All Options view:

```
/I"..\..\windows\include" /Zi /nologo /W1 /WX- /O2 /Ob2 /Oi /Ot /Oy- /D "PCRE_STATIC"␣
↪/D "_WINDLL" /D "_MBCS" /Gm- /MD /GS- /fp:precise /Zc:wchar_t /Zc:forScope /Fp
↪"Release\libyara.pch" /Fa"Release\" /Fo"Release\" /Fd"Release\vc100.pdb" /Gd /TC /wd
↪"4996" /analyze- /errorReport:queue
```

The *Linker* All Options view:

```
/OUT:".\yara\tags\yara-1.6.0\windows\libyara\Release\libyara.dll" /NOLOGO /LIBPATH:"..
↪\lib" /LIBPATH:".\yara\tags\yara-1.6.0\windows\libyara\Release\" /DLL "pcre32.lib"
↪"kernel32.lib" "user32.lib" "gdi32.lib" "winspool.lib" "comdlg32.lib" "advapi32.lib
↪" "shell32.lib" "ole32.lib" "oleaut32.lib" "uuid.lib" "odbc32.lib" "odbccp32.lib" /
↪MANIFEST /ManifestFile:"Release\libyara.dll.intermediate.manifest" /ALLOWISOLATION /
↪MANIFESTUAC:"level='asInvoker' uiAccess='false'" /PDB:".\yara\tags\yara-1.6.
↪0\windows\libyara\Release\libyara.pdb" /PGD:".\yara\tags\yara-1.6.
↪0\windows\libyara\Release\libyara.pgd" /TLBID:1 /DYNAMICBASE /NXCOMPAT /MACHINE:X86␣
↪/ERRORREPORT:QUEUE
```

Finally, to export the functions in the libyara.dll you need to ensure that each export function `includes/yara.h` has a `__declspec(dllexport)` defined before it:

```
>>>yara.h<<<
 __declspec(dllexport) RULE*           lookup_rule(RULE_LIST* rules, const char*␣
↪identifier, NAMESPACE* ns);
 __declspec(dllexport) STRING*         lookup_string(STRING* string_list_head,␣
↪const char* identifier);
 __declspec(dllexport) TAG*            lookup_tag(TAG* tag_list_head, const char*␣
↪identifier);
 __declspec(dllexport) META*           lookup_meta(META* meta_list_head, const␣
↪char* identifier);
 __declspec(dllexport) VARIABLE*       lookup_variable(VARIABLE* _list_head, const␣
↪char* identifier);
 __declspec(dllexport) void            yr_init();
 __declspec(dllexport) YARA_CONTEXT*   yr_create_context();
 __declspec(dllexport) void            yr_destroy_context(YARA_CONTEXT* context);
 __declspec(dllexport) int             yr_calculate_rules_weight(YARA_CONTEXT*␣
↪context);
 __declspec(dllexport) NAMESPACE*      yr_create_namespace(YARA_CONTEXT* context,␣
↪const char* name);
 __declspec(dllexport) int             yr_define_integer_variable(YARA_CONTEXT*␣
↪context, const char* identifier, size_t value);
```

```
__declspec(dllexport) int                 yr_define_boolean_variable(YARA_CONTEXT*␣
↪context, const char* identifier, int value);
__declspec(dllexport) int                 yr_define_string_variable(YARA_CONTEXT*␣
↪context, const char* identifier, const char* value);
__declspec(dllexport) int                 yr_undefine_variable(YARA_CONTEXT* context,␣
↪const char* identifier);
__declspec(dllexport) char*               yr_get_current_file_name(YARA_CONTEXT*␣
↪context);
__declspec(dllexport) int                 yr_push_file_name(YARA_CONTEXT* context,␣
↪const char* file_name);
__declspec(dllexport) void                yr_pop_file_name(YARA_CONTEXT* context);
__declspec(dllexport) int                 yr_compile_file(FILE* rules_file, YARA_
↪CONTEXT* context);
__declspec(dllexport) int                 yr_compile_string(const char* rules_string,␣
↪YARA_CONTEXT* context);
__declspec(dllexport) int                 yr_scan_mem(unsigned char* buffer, size_t␣
↪buffer_size, YARA_CONTEXT* context, YARACALLBACK callback, void* user_data);
__declspec(dllexport) int                 yr_scan_file(const char* file_path, YARA_
↪CONTEXT* context, YARACALLBACK callback, void* user_data);
__declspec(dllexport) int                 yr_scan_proc(int pid, YARA_CONTEXT* context,␣
↪YARACALLBACK callback, void* user_data);
__declspec(dllexport) char*               yr_get_error_message(YARA_CONTEXT* context,␣
↪char* buffer, int buffer_size);
__declspec(dllexport) void                yr_free_matches(YARA_CONTEXT* context);
```

Building for OS X Mountain Lion

Install Homebrew and install the following packages:

```
brew install libtool pcre bison automake autoconf svn
```

Patch libyara/configure.ac with the following:

```
>>>libyara/configure.ac<<<
+ m4_pattern_allow([AM_PROG_AR])
+ AM_PROG_AR
```

Reconfigure the auto build tool chain:

```
autoreconf -fiv
```

Due to a bug in the auto config files (somewhere) replace the generated libyara/libtool with:

```
rm libyara/libtool
ln -s /usr/local/Cellar/libtool/2.4.2/bin/glibtool libyara/libtool
```

Copy and rename the dynamic link library:

```
cp ./libyara/.libs/libyara.0.dylib <DESTPATH>/libyara.so
```

Bundling libyara shared library files

You can add your own libyara.dll/so files to the `.libs/` folder before running `python setup.py install`

Windows:

```
./libs/windows/x86_64/libyara.dll
./libs/windows/x86/libyara.dll
```

Linux:

```
./libs/linux/x86_64/libyara.so
./libs/linux/x86/libyara.so
```

OS X:

```
./libs/darwin/x86_64/libyara.so
```

Alternatively you can install your libyara files in the correct place such that `libyara_wrapper` can find them.

i.e:

```
Windows:
   <python install dir>\DLLs   (or sys.prefix + 'DLLs')
Linux:
   <python env usr root>/lib    (or sys.prefix + 'lib'
```

CHAPTER 2

Reference

yara.scan — Thread pool execution of rules matching

This module is responsible for implementing the base Scanner type and various extensions to meet different scanning requirements.

Scanner

Scanner([rules_rootpath,whitelist,blacklist,rule_filepath,
thread_pool,
externals])

This is the base Scanner class which initialises and aggregates a Rules class to perform match jobs against. It has the responsibility of managing a job queue and result queue and sets up the interface required for child class Scanner instances.

Scanner implements the iter protocol which yields scan results as they complete. To enable more efficient scanning, Scanner deploys a thread pool for concurrent scanning and manages its execution through its internal job queues. Once a job completes, the job tag id and the results are returned through the dequeue function or yielded during iteration.

PathScanner

class yara.scan.**PathScanner**([*args*, *recurse_dirs*, *path_end_include*, *path_end_exclude*, *path_contains_include*, *path_contains_exclude*, *rules_rootpath*, ***scanner_kwargs*])

PathScanner extends the Scanner class to enable simple queuing of filepaths found in the file system. It defines an exclude_path algorithm which utilises the path include exclude. *PathScanner* has a paths property which is an interator for yielding the filepaths it discovers based on the various constraints.

The following example demonstrates how *PathScanner* can be operated:

```
# Recursively scan all subdirectories from the path '.'
for path, result in PathScanner(args=['.']):
    print("%s : %s" % (path, result))
```

FileChunkScanner

class `yara.scan.`**`FileChunkScanner`**`(`[*file_chunk_size*, *file_readahead_limit*, ***path_scanner_kwargs*]`)`

`FileChunkScanner` extends *`PathScanner`* and defines a way to reads chunks of data from filepaths choosen by *`PathScanner`* and enqueue `Rules`.match_data jobs.

PidScanner

class `yara.scan.`**`PidScanner`**`(`[*args*, ***scanner_kwargs*]`)`

`PidScanner` ...

`yara.cli` — A command line YARA rules scanning utility

This module is responsible for implementing the CLI that allows users to rapidly execute their yara signatures against file(s) and pid(s).

See *How to scan* for more details.

`yara.rules` — YARA namespaces, compilation, and matching

Compiles a YARA rules files into a thread safe Rules object ready for matching.

Features:

- Provides a thread safe yara context manager.
- Detailed control over the loading of multiple YARA rules files into a
- single context.
- A C-like preprocessor for yar files. Allows for #ifdef #ifndef etc.

Key differences to yara-python.c:

- Results returned from a `Rules.match(_??)` function are stored in a dict of `{namespace:[match, ...]}`...
- When a callback hander is passed into a `Rules.match(_??)` function, the match function will return an empty dict. It is assumed that the callback handler will retain the match objects that it cares about.
- The match dict inside of a dict returned from a `Rules.match(_??)` function no longer contain the namespace (namespace is the key used to reference the match dict).

Compatibility with yara-python.c

- This module contains an equivalent `compile()` function
- The Rules object contains an equivalent `match()` function

- Match objects passed into the registered callback handler are the equivalent

Rules

class `yara.rules.`**`Rules`**(*paths={}*, *defines={}*, *include_path=[]*, *strings=[]*, *externals={}*, *fast_match=False*)

Rules manages the seamless construction of a new context per thread and exposes libyara's match capability.

`__init__`(*paths={}*, *defines={}*, *include_path=[]*, *strings=[]*, *externals={}*, *fast_match=False*)

Defines a new yara context with specified yara sigs

Options: paths - {namespace:rules_path,...} include_path - a list of paths to search for given #include

directives.

defines - key:value defines for the preprocessor. Sub in strings or macros defined in your rules files.

strings - [(namespace, filename, rules_string),...] externals - define boolean, integer, or string variables

{var:val,...}

fast_match - enable fast matching in the YARA context

Note: namespace - defines which namespace we're building our rules under rules_path - path to the .yar file filename - filename which the rules_string came from rules_string - the text read from a .yar file

`match`(*filepath=None*, *pid=None*, *data=None*, ***match_kwargs*)

Match on one of the following: pid= filepath= or data= Require one of the following:

filepath - filepath to match against pid - process id data - filepath to match against

Options: externals - define boolean, integer, or string variables callback - provide a callback function which will get called with

the match results as they comes in.

Note #1: If callback is set, the Rules object doesn't bother storing the match results and this func will return []... The callback hander needs to deal with individual matches.

Note #2: The callback can abort the matching sequence by returning a CALLBACK_ABORT or raising a StopIteration() exception. To continue, a return object of None or CALLBACK_CONTINUE is required.

Functionally equivalent to (yara-python.c).match

`match_data`(*data*, *externals={}*, *callback=None*)

Match data against the compiled rules Required argument:

data - filepath to match against

Options: externals - define boolean, integer, or string variables callback - provide a callback function which will get called with

the match results as they comes in.

Note #1: If callback is set, the Rules object doesn't bother storing the match results and this func will return []... The callback hander needs to deal with individual matches.

Note #2: The callback can abort the matching sequence by returning a CALLBACK_ABORT or raising a StopIteration() exception. To continue, a return object of None or CALLBACK_CONTINUE is required.

Return a dictionary of {"namespace":[match1,match2,...]}

match_path (*filepath*, *externals={}*, *callback=None*)

Match a filepath against the compiled rules Required argument:

filepath - filepath to match against

Options: externals - define boolean, integer, or string variables callback - provide a callback function which will get called with

the match results as they comes in.

Note #1: If callback is set, the Rules object doesn't bother storing the match results and this func will return []... The callback hander needs to deal with individual matches.

Note #2: The callback can abort the matching sequence by returning a CALLBACK_ABORT or raising a StopIteration() exception. To continue, a return object of None or CALLBACK_CONTINUE is required.

Return a dictionary of {"namespace":[match1,match2,...]}

match_proc (*pid*, *externals={}*, *callback=None*)

Match a process memory against the compiled rules Required argument:

pid - process id

Options: externals - define boolean, integer, or string variables callback - provide a callback function which will get called with

the match results as they comes in.

Note #1: If callback is set, the Rules object doesn't bother storing the match results and this func will return []... The callback hander needs to deal with individual matches.

Note #2: The callback can abort the matching sequence by returning a CALLBACK_ABORT or raising a StopIteration() exception. To continue, a return object of None or CALLBACK_CONTINUE is required.

Return a dictionary of {"namespace":[match1,match2,...]}

yara.rules.load_rules()

yara.rules.**load_rules**(*rules_rootpath='/home/docs/checkouts/readthedocs.org/user_builds/yara-ctypes/envs/latest/local/lib/python2.7/site-packages/yara-1.7.7-py2.7.egg/yara/rules', blacklist=[], whitelist=[], include_path=['/home/docs/checkouts/readthedocs.org/user_builds/yara-ctypes/envs/latest/bin', '/usr/local/sbin', '/usr/local/bin', '/usr/sbin', '/usr/bin', '/sbin', '/bin'], **rules_kwargs*)

A simple way to build a complex yara Rules object with strings equal to [(namespace:filepath:source),...]

YARA rules files found under the rules_rootpath are loaded based on the exclude namespace blacklist or include namespace whitelist.

i.e. Where rules_rootpath = './rules' which contained:

./rules/hbgary/libs.yar ./rules/hbgary/compression.yar ./rules/hbgary/fingerprint.yar

The resultant Rules object would contain the following namespaces: hbgary.libs hbgary.compression hbgary.fingerprint

Optional YARA rule loading parameters: rules_rootpath - root dir to search for YARA rules files blacklist - namespaces "starting with" to exclude whitelist - namespaces "starting with" to include

Rule options: externals - define boolean, integer, or string variables {var:val,...} fast_match - enable fast matching in the YARA context

`yara.rules.compile()`

`yara.rules.`**`compile`**(*filepath=None*, *source=None*, *fileobj=None*, *filepaths=None*, *sources=None*, ***rules_kwargs*)

Compiles a YARA rules file and returns an instance of class Rules

Require one of the following: filepath - str object containing a YARA rules filepath source - str object containing YARA source fileobj - a file object containing a set of YARA rules filepaths - {namespace:filepath,...} sources - {namespace:source_str,...}

Rule options: externals - define boolean, integer, or string variables {var:val,...} fast_match - enable fast matching in the YARA context

Functionally equivalent to (yara-python.c).compile

`yara.libyara_wrapper` — ctypes wrapper for libyara

This module is responsible for wrapping the libyara dynamic library various exported functions. Using ctypes, it replicates the yara data structures and exported functions.

See *How to build* for library details.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

y

yara.libyara_wrapper, 17
yara.rules, 14
yara.scan, 13

Index

Symbols

__init__() (yara.rules.Rules method), 15

C

compile() (in module yara.rules), 17

F

FileChunkScanner (class in yara.scan), 14

L

load_rules() (in module yara.rules), 16

M

match() (yara.rules.Rules method), 15
match_data() (yara.rules.Rules method), 15
match_path() (yara.rules.Rules method), 16
match_proc() (yara.rules.Rules method), 16

P

PathScanner (class in yara.scan), 13
PidScanner (class in yara.scan), 14

R

Rules (class in yara.rules), 15

Y

yara.libyara_wrapper (module), 17
yara.rules (module), 14
yara.scan (module), 13, 14