Документация k YARA

Release 3.8.1

Victor M. Alvarez, перевод: Дроботун Евгений

Apr 24, 2020

Оглавление

The pattern matching swiss knife for malware researchers (and everyone else).

Release 3.8.1

Victor M. Alvarez

Dec 19, 2018

ГЛАВА 1

Документация к YARA (перевод на русский)

Дроботун Евгений (drobotun@xakep.ru)

1.1 Начало работы

YARA - это мультиплатформенная программа, работающая в операционных системах Windows, Linux и Mac OS X. Вы можете найти последнюю версию YARA по адресу.

1.1.1 Компиляция и установка YARA

Загрузите исходный архив и подготовьтесь к его компиляции:

```
tar -zxf yara-3.8.1.tar.gz
cd yara-3.8.1
./bootstrap.sh
```

Убедитесь, что в вашей системе установлены automake, libtool, make и gcc. Пользователи Ubuntu и Debian могут использовать:

```
sudo apt-get install automake libtool make gcc
```

Если вы планируете изменить исходный код YARA, вам также могут понадобиться flex и bison для генерации лексеров и парсеров:

```
sudo apt-get install flex bison
```

Скомпилируйте и установите YARA стандартным способом:

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

Запустите проверку, чтобы убедиться, что все нормально:

```
make check
```

Некоторые функции YARA зависят от библиотеки OpenSSL. Эти функции будут включены, только если в системе установлена библиотека OpenSSL. Если данная библиотека отсутствует, YARA будет работать нормально, но вы не сможете использовать отключенные функции.

Сценарий configure автоматически определит, установлен OpenSSL или нет. Если вы хотите применить зависимые функции OpenSSL, вы должны передать --with-crypto скрипту configure. Пользователи Ubuntu и Debian могут использовать sudo apt-get install libssl-dev для установки библиотеки OpenSSL.

По умолчанию в YARA не компилируются следующие модули:

- cuckoo
- magic
- dotnet

Если вы планируете использовать эти модули, вы должны передать соответствующие аргументы --enable-<module name> в скрипт configure.

Например:

```
./configure --enable-cuckoo
./configure --enable-magic
./configure --enable-dotnet
./configure --enable-cuckoo --enable-magic --enable-dotnet
```

Модули обычно зависят от внешних библиотек и, в зависимости от модулей, которые вы решите установить, вам могут понадобиться следующие библиотеки:

- cuckoo: Зависит от библиотеки Jansson для разбора JSON-строк. Некоторые версии Ubuntu и Debian уже включают пакет с именем libjansson-dev, однако если sudo apt-get install libjansson-dev у вас не работает, то можно загрузить исходный код из этого репозитория.
- magic: Зависит от библиотеки libmagic, используемой стандартным программным файлом Unix. Операционные системы Ubuntu, Debian и CentOS обычно включают в себя пакет libmagic-dev. Исходный код можно найти здесь.

Установка в Windows

Скомпилированные двоичные файлы для Windows в 32 и 64-разрядных вариантах можно найти по ссылке. Просто скачайте нужную версию, распакуйте архив и сохраните файлы yara.exe и yarac.exe в нужном месте на диске.

Для установки расширения yara-python загрузите и выполните установщик, соответствующий используемой версии Python.

Установка в Mac OS X с Homebrew

Чтобы установить YARA с помощью Homebrew, просто введите brew install yara.

Установка yara-python

Если вы планируете использовать YARA из ваших скриптов Python, вам необходимо установить расширение yara-python. Это расширение можно найти обратившись к https://github.com/VirusTotal/yara-python.

1.1.2 Запуск YARA первый раз

Теперь, когда вы установили YARA, вы можете написать очень простое правило и использовать инструмент командной строки для сканирования некоторых файлов:

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

Пусть вас не удивляет повторяющиеся my_first_rule в аргументах YARA, в данном случае для сканирования передается тот же файл, что и файл с правилом. Вы можете передать любой файл, который хотите проверить (второй аргумент).

Если все пойдет хорошо, вы должны получить следующий результат:

```
dummy my_first_rule
```

Это означает, что файл my_first_rule соответствует правилу с именем dummy.

Если вы получаете ошибку такого вида:

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

Это означает, что загрузчик не находит библиотеку libyara, которая находится в /usr/local/lib. В некоторых вариантах Linux загрузчик по умолчанию не ищет библиотеки по этому пути, поэтому мы должны указать ему сделать это, добавив /usr/local/lib в файл конфигурации загрузчика /etc/ld.so.conf:

```
sudo sh -c 'echo "/usr/local/lib" >> /etc/ld.so.conf'
sudo ldconfig
```

1.2 Написание правил YARA

Правила YARA достаточно просто писать. Синтаксис правил напоминает язык программирования C. Вот самое простое правило, которое можно написать для YARA и которое абсолютно ничего не делает:

```
rule dummy
{
        condition:
                false
}
```

Каждое правило в YARA начинается с ключевого слова rule, за которым следует идентификатор правила. Идентификаторы должны соответствовать лексическим соглашениям языка программирования C, они могут содержать любые буквенно-цифровые символы и символы подчеркивания, но при этом первый символ не может быть цифровым. Идентификаторы правил чувствительны к регистру и не могут превышать длину в 128 символов. Следующие ключевые слова зарезервированы и не могут использоваться в качестве идентификатора:

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

Правила обычно состоят из двух разделов: определение строк (strings) и условие (condition). Раздел trings может быть опущен, если правило не зависит от какой-либо строки, раздел condition должен присутствовать в любом правиле. В разделе strings определяются строки, которые будут частью правила. Каждая строка имеет идентификатор, состоящий из символа $, за которым следует последовательность буквенно-цифровых символов и символов подчеркивания, эти идентификаторы могут использоваться в разделе condition для ссылки на соответствующую строку. Строки могут быть определены в текстовой или шестнадцатеричной форме, как показано в следующем примере:

```
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
}
```

Текстовые строки заключаются в двойные кавычки, как и в языке C. Шестнадцатеричные строки заключены в фигурные скобки, и они состоят из последовательности шестнадцатеричных чисел, которые могут быть записаны непрерывно или разделяться пробелами.

В шестнадцатеричных строках не допускается использование десятичных чисел.

В разделе condition содержится логика правила. Этот раздел должен содержать логическое выражение, которое показывает, при каких обстоятельствах файл или процесс удовлетворяет правилу или нет. Как правило, условие будет ссылаться на ранее определенные строки с помощью их идентификаторов. В этом контексте идентификатор строки действует как логическая переменная, которая вычисляется как true, если строка была найдена в памяти файла или процесса, или false, в противном случае.

1.2.1 Комментарии

Вы можете добавлять комментарии к своим правилам YARA так же, как если бы это был исходный файл C, как однострочный, так и многострочный стиль C комментарии поддерживаются.

```
/*
        это многострочный комментарий ...
*/
rule CommentExample // ... это однострочный комментарийt
{
        condition:
                false // просто фиктивное правило
}
```

1.2.2 Строки

В YARA применяется три типа строк: шестнадцатеричные строки, текстовые строки и регулярные выражения. Шестнадцатеричные строки используются для определения последовательности байтов, в то время как текстовые строки и регулярные выражения используются для определения части текста, содержащегося в файле или процессе. При этом текстовые строки и регулярные выражения могут также использоваться для представления байтов посредством escape-последовательностей, как будет показано ниже.

Шестнадцатеричные строки

Шестнадцатеричные строки допускают три специальные конструкции, которые делают их более гибкими: подстановочные знаки, переходы и альтернативы. Подстановочные знаки – это просто заполнители, которые вы можете поместить в строку, указывающую, что некоторые байты неизвестны, и они должны соответствовать чему-либо. Символ-заполнитель – знак вопроса (?). Вот пример шестнадцатеричной строки с подстановочными знаками:

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

Как показано в примере, подстановочным знаком можно определить не только байт целиком, но и его часть (старшую или младшую тетраду).

Подстановочные знаки полезны при определении строк, содержимое которых может изменяться, но при этом размер их известен, однако это не всегда так. В некоторых случаях может потребоваться определить строки с фрагментами переменного содержимого и длины. В таких ситуациях вы можете использовать переходы вместо подстановочных знаков:

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

В приведенном выше примере у нас есть пара чисел, заключенный в квадратные скобки и разделенных дефисом, это переход. Он показывает, что любая произвольная последовательность от 4 до 6 байт может занимать позицию перехода. Любая из следующих строк будет соответствовать шаблону:

```
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
```

Любой переход [X-Y] должен удовлетворять условию 0 <= X <= Y. В предыдущих версиях YARA и X, и Y могли принимать значения не более 256, но начиная с YARA 2.0 для X и Y это ограничение снято.

Например:

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

Такая запись является недопустимой:

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

Если нижняя и верхняя границы равны, вы можете написать одно число, заключенное в скобки, например:

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

Приведенная выше строка эквивалентна обоим из них:

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

Начиная с YARA 2.0 вы также можете использовать неограниченные переходы:

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

Первый означает [10-бесконечно], второй означает [0-бесконечно].

Есть также ситуации, в которых вы можете предоставить различные альтернативы для данного фрагмента шестнадцатеричной строки. В таких ситуациях можно использовать синтаксис, напоминающий регулярное выражение:

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

Это правило будет соответствовать любому файлу, содержащему F42362B445 или F4235645.

Можно выразить и более двух альтернатив. Фактически, нет никаких ограничений ни на количество альтернативных последовательностей, ни на их размер.

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

Как можно увидеть в приведенном выше примере, строки, содержащие подстановочные символы можно использовать в рамках альтернативных последовательностей.

Текстовые строки

Как было показано ранее, текстовые строки обычно определяются следующим образом:

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

Это самый простой случай: строка в кодировке ASCII с учетом регистра. Однако текстовые строки могут сопровождаться некоторыми полезными модификаторами, которые изменяют способ интерпре-

тации строки. Эти модификаторы добавляются в конце строки, разделенные пробелами, как будет показано ниже.

Текстовые строки могут также содержать следующее подмножество escape-последовательностей, доступных на языке Си:

- \" - Двойная кавычка
- \\ - Обратный слэш
- \t - Горизонтальная табуляция
- \n - Новая строка
- \xdd - Любой байт в шестнадцатеричной нотации

Регистро-независимые строки

Текстовые строки в YARA чувствительны к регистру по умолчанию, однако вы можете превратить свою строку в режим без учета регистра, добавив модификатор nocase в конце определения строки в той же строке:

```
rule CaseInsensitiveTextExample
{
        strings:
                $text_string = "foobar" nocase
        condition:
                $text_string
}
```

С модификатором nocase строка foobar будет соответствовать Foobar, FOOBAR и fOoBaR. Этот модификатор может использоваться совместно с любым другим модификатором.

Расширенные строки

Модификатор wide может использоваться для поиска строк, закодированных двумя байтами на символ, что типично для многих исполняемых бинарных файлов. В приведенном ниже примере строка “Borland” кодируется как два байта на символ:

```
rule WideCharTextExample1
{
        strings:
                $wide_string = "Borland" wide
        condition:
                $wide_string
}
```

Однако необходимо иметь в виду, что этот модификатор просто чередует коды ASCII-символов в строке с нулями, он не поддерживает строки UTF-16, содержащие неанглийские символы.

Если вы хотите найти строки в обоих форматах (ASCII и расширенном), вы можете использовать модификатор ascii в сочетании с wide, независимо от того, в каком порядке они появляются.

```
rule WideCharTextExample2
{
        strings:
                $wide_and_ascii_string = "Borland" wide ascii
```

(continues on next page)

(continued from previous page)

```
        condition:
                $wide_and_ascii_string
}
```

Модификатор ascii может быть использован отдельно, без сопутствующего модификатора wide, при этом писать его не обязательно, так как в отсутствие модификатора wide строка по умолчанию считается ASCII.

XOR-строки

Модификатор xor может использоваться для поиска строк, к каждому байту которой применена операция “исключающее ИЛИ” (сложение по модулю 2) с каким-либо произвольным байтом.

Следующее правило будет искать строки, полученные при применении к строке "This program cannot" операции “исключающее ИЛИ” с любым произвольным байтом:

```
rule XorExample1
{
        strings:
                $xor_string = "This program cannot" xor
        condition:
                $xor_string
}
```

Приведенное выше правило логически эквивалентно правилу:

```
rule XorExample2
{
        strings:
                $xor_string_00 = "This program cannot"
                $xor_string_01 = "Uihr!qsnfs`l!b`oonu"
                $xor_string_02 = "Vjkq\"rpmepco\"acllmv"
                // Повторить для каждого байта операции xor
        condition:
                any of them
}
```

Вы также можете комбинировать xor модификатор с wide, ascii и nocase модификаторами. Например, для поиска расширенной и ASCII-версии строки после применения к ней “исключающего ИЛИ” следует использовать:

```
rule XorExample3
{
        strings:
                $xor_string = "This program cannot" xor wide ascii
        condition:
                $xor_string
}
```

Модификатор xor применяется после каждого другого модификатора. Это означает, что использование xor и wide вместе приводит к применению xor к чередующимся нулевым байтам. Например, следующие два правила логически эквивалентны:

```
rule XorExample3
{
        strings:
                $xor_string = "This program cannot" xor wide
        condition:
                $xor_string
}

rule XorExample4
{
        strings:
                $xor_string_00 = "T\x00h\x00i\x00s\x00 \x00p\x00r\x00o\x00g\x00r\x00a\x00m\x00
↪\x00c\x00a\x00n\x00n\x00o\x00t\x00"
                $xor_string_01 = "U\x01i\x01h\x01r\x01!\x01q\x01s\x01n\x01f\x01s\x01`\x01l\x01!
↪\x01b\x01`\x01o\x01o\x01n\x01u\x01"
                $xor_string_02 = "V\x02j\x02k\x02q\x02\"\x02r\x02p\x02m\x02e\x02p\x02c\x02o\x02\
↪"\x02a\x02c\x02l\x02l\x02m\x02v\x02"
                // Повторить для каждого байта операции xor
        condition:
                any of them
}
```

Поиск полных слов

Другим модификатором, который может быть применен к текстовым строкам, является fullword. Этот модификатор гарантирует, что строка будет соответствовать, только если она появляется в файле (или процессе), разделенном не буквенно-цифровыми символами. Например, строка domain, если она определена как полное слово, не соответствует www.mydomain.com, но при этом соответствует www.my-domain.com и www.domain.com.

Регулярные выражения

Регулярные выражения являются одной из самых мощных функций YARA. Они определяются так же, как и текстовые строки, но заключаются в косые черты вместо двойных кавычек, как в языке программирования Perl.

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

Регулярные выражения могут также сопровождаться модификаторами nocase, ascii, wide и fullword, как и в текстовых строках. Семантика этих модификаторов одинакова в обоих случаях.

В предыдущих версиях YARA для сопоставления регулярных выражений использовались внешние библиотеки, такие как PCRE и RE2, но начиная с версии 2.0 YARA использует собственный механизм регулярных выражений. Он реализует большинство функций, входящих в PCRE, за исключением некоторых из них, таких как группы захвата, классы символов POSIX и обратные ссылки.

Регулярные выражения YARA распознают следующие метасимволы:

- \ - Экранирует следующие метасимволы
- ^ - Показывает начало файла
- $ - Показывает конец файла
- | - Выбор альтернатив
- () - Группирование
- [] - Класс символов

Также могут использоваться следующие квантификаторы:

- * - 0 или более раз
- + - 1 или более раз
- ? - 0 или 1 раз
- {n} - Ровно n раз
- {n,} - Не менее n раз
- {,m} - Не более m раз
- {n,m} - От n до m раз

Все эти квантификаторы имеют “ленивый” вариант работы, который обозначается знаком вопроса ?:

- *? - 0 или более раз в “ленивом” режиме
- +? - 1 или более раз в “ленивом” режиме
- ?? - 0 или 1 раз в “ленивом” режиме
- {n}? - Ровно n раз в “ленивом” режиме
- {n,}? - Не менее n раз в “ленивом” режиме
- {,m}? - Не более m раз в “ленивом” режиме
- {n,m}? - От n до m раз в “ленивом” режиме

Могут использоваться следующие escape-последовательности:

- \t - Tab (HT, TAB)
- \n - New line (LF, NL)
- \r - Return (CR)
- \f - Form feed (FF)
- \a - Alarm bell
- \xNN - Символ, порядковым номером которого является данное шестнадцатеричное число

Классы символов:

- \w - Словарные символы (буквенно-цифровые и “_”)
- \W - Не словарные символы
- \s - Пробел
- \S - Не пробельные символы
- \d - Символы десятичных цифр
- \D - Не цифровые символы

Начиная с версии 3.3.0 также возможно применение:

- \b - Граница слова
- \B - Совпадает на границе слова

1.2.3 Условия

Условия - это не что иное, как логические выражения, которые можно найти во всех языках программирования, например оператор if. Они могут содержать типичные булевы операторы and, or, и not, и реляционные операторы >=, <=, <, >, == и !=. Кроме того, арифметические операторы (+, -, *, \, %) и побитовые операторы (&, |, <<, >>, ~, ^) могут использоваться для числовых выражений.

Строковые идентификаторы могут также использоваться в условии, действуя как булевы переменные, значение которых зависит от наличия или отсутствия связанной строки в файле.

```
rule Example
{
        strings:
                $a = "text1"
                $b = "text2"
                $c = "text3"
                $d = "text4"
        condition:
                ($a or $b) and ($c or $d)
}
```

Подсчет строк

Иногда нам нужно знать не только, присутствует ли определенная строка или нет, но и сколько раз строка появляется в файле или памяти процесса. Число вхождений каждой строки представлено переменной, имя которой строковый идентификатор, но с символом # вместо символа $. Например:

```
rule CountExample
{
        strings:
                $a = "dummy1"
                $b = "dummy2"
        condition:
                #a == 6 and #b > 10
}
```

Это правило соответствует любому файлу или процессу, содержащему строку $a ровно шесть раз и более десяти вхождений строки $b.

Смещение строк или виртуальный адрес

В большинстве случаев, когда строковый идентификатор используется в условии, мы хотим знать, находится ли связанная строка где-либо в файле или памяти процесса, но иногда нам нужно знать, находится ли строка в некотором определенном смещении в файле или в некотором виртуальном адресе в адресном пространстве процесса. В таких ситуациях оператор at- это то, что нам нужно. Этот оператор используется, как показано в следующем примере:

```
rule AtExample
{
        strings:
                $a = "dummy1"
                $b = "dummy2"
        condition:
                $a at 100 and $b at 200
}
```

Выражение $a at 100 в приведенном выше примере истинно только в том случае, если строка $a находится со смещением 100 в файле (или по виртуальному адресу 100, если применяется к запущенному процессу). Строка $b должна находится по смещению 200. Обратите внимание, что оба смещения являются десятичными, однако шестнадцатеричные числа также можно использовать, добавив префикс 0x перед числом, как в языке программирования C, что очень удобно при написании виртуальных адресов. Также обратите внимание на более высокий приоритет оператора at над and.

В то время как оператор at позволяет искать строку с некоторым фиксированным смещением в файле или виртуальном адресе в пространстве памяти процесса, оператор in позволяет искать строку в диапазоне смещений или адресов.

```
rule InExample
{
        strings:
                $a = "dummy1"
                $b = "dummy2"
        condition:
                $a in (0..100) and $b in (100..filesize)
}
```

В приведенном выше примере строка $a должна быть найдена со смещением от 0 до 100, а строка $b - со смещением от 100 до конца файла. Опять же, по умолчанию, числа десятичные.

Вы также можете получить смещение или виртуальный адрес i-го вхождения строки $a с помощью @a[i]. Первый индекс - единица, поэтому первое вхождение будет @a[1] второе @a[2] и так далее. Если указать индекс, превышающий число вхождений строки, результатом будет значение NaN (Not A Number).

Длина совпадений

Для многих регулярных выражений и шестнадцатеричных строк, содержащих переходы, длина совпадений является переменной. Если у вас есть регулярное выражение /fo*/ строки "fo", "foo" и "fooo" могут быть совпадениями, при этом все они разной длины.

Вы можете использовать длину совпадений как часть вашего условия с помощью символа ! перед строковым идентификатором, также как используется символ @ для смещения. !a[1] - длина первого совпадения $a, !a[2] - длина второго совпадения и так далее. !a является сокращенной формой !a[1].

Размер файла

Строковые идентификаторы не являются единственными переменными, которые могут отображаться в условии (на самом деле, правила могут быть определены без определения строки, как будет показано ниже), есть и другие специальные переменные, которые могут быть использованы. Одна из этих специальных переменных - переменная filesize, которая содержит, как указывает ее имя, размер сканируемого файла. Размер выражается в байтах.

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

Предыдущий пример также демонстрирует использование постфикса KB. Этот постфикс при присоединении к числовой константе автоматически умножает значение константы на 1024. Постфикс MB можно использовать для умножения значения на 2^20. Оба постфикса можно использовать только с десятичными константами.

Использование filesize имеет смысл только тогда, когда правило применяется к файлу. Если правило применяется к запущенному процессу, оно всегда будет получать на выходе значение false, потому что filesize в данном случае не имеет смысла.

Точка входа исполняемого файла

Другой специальной переменной, которая может использоваться в правиле, является entrypoint. Если файл является Portable Executable (PE) или Executable and Linkable Format (ELF), эта переменная содержит смещение точки входа исполняемого файла в случае сканирования файла. Если мы сканируем запущенный процесс, точка входа будет содержать виртуальный адрес точки входа основного исполняемого файла. Обычно эта переменная используется для поиска некоторого шаблона в точке входа для обнаружения упаковщиков или простых файловых инфекторов.

```
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)
}
```

Наличие переменной entrypoint в правиле означает, что только файлы PE или ELF могут удовлетворять этому правилу. Если файл не является PE или ELF, любое правило, использующее эту переменную, получает значение false.

> Warning: Переменная entrypoint устарела, вы должны использовать эквивалентную переменную pe.entry_point из модуля PE. Начиная с YARA 3.0 при использовании entrypoint вы получите предупреждение. Данная переменная будет удалена из последующих версий.

Доступ к данным на заданной позиции

Есть много ситуаций, в которых вы можете записать условия, которые зависят от данных, хранящихся по определенному смещению в файле или по виртуальному адресу процесса, в зависимости от того,

сканируем мы файл или запущенный процесс. В таких случаях можно использовать одну из следующих функций для чтения данных из файла с заданным смещением:

```
int8(смещение или виртуальный адрес)
int16(смещение или виртуальный адрес)
nt32(смещение или виртуальный адрес)

uint8(смещение или виртуальный адрес)
uint16(смещение или виртуальный адрес)
uint32(смещение или виртуальный адрес)

int8be(смещение или виртуальный адрес)
int16be(смещение или виртуальный адрес)
int32be(смещение или виртуальный адрес)

uint8be(смещение или виртуальный адрес)
uint16be(смещение или виртуальный адрес)
uint32be(смещение или виртуальный адрес)
```

Функции intXX считывают 8, 16 и 32-разрядные целые числа со знаком по указанному смещению или виртуальному адресу, а функции uintXX - целые числа без знака. Как 16, так и 32-разрядные целые числа считываются в little-endian формате. Если вы хотите прочитать целое число в big-endian формате, используйте соответствующую функцию, заканчивающуюся на be. В качестве значения смещения или виртуального адреса может быть любое выражение, возвращающее целое число без знака, включая возвращаемое значение одной из функций uintXX. В качестве примера рассмотрим правило для определения PE-файлов:

```
rule IsPE
{
        condition:
                // MZ-сигнатура по смещению 0 и ...
                uint16(0) == 0x5A4D and
                // ... PE-сигнатура по смещению 0x3C в MZ-заголовке
                uint32(uint32(0x3C)) == 0x00004550
}
```

Наборы строк

Есть обстоятельства, в которых надо указать, что файл должен содержать определенное количество строк из заданного набора. Не все строки из наборе должны присутствовать в файле, но, по крайней мере некоторые из них должны содержаться в файле. В этих ситуациях можно использовать оператор of.

```
rule OfExample1
{
        strings:
                $a = "dummy1"
                $b = "dummy2"
                $c = "dummy3"
        condition:
                2 of ($a,$b,$c)
}
```

Это правило требует, чтобы по крайней мере две строки из набора ($a, $b, $c) присутствовали в файле, но не имеет значения, какие две из них. Конечно, при использовании этого оператора, число до оператора должно быть меньше или равно количеству строк в наборе.

Элементы набора могут быть явно перечислены, как в предыдущем примере, или могут быть указаны с помощью подстановочных символов. Например:

```
rule OfExample2
{
        strings:
                $foo1 = "foo1"
                $foo2 = "foo2"
                $foo3 = "foo3"
        condition:
                2 of ($foo*) // эквивалент для выражения 2 of ($foo1,$foo2,$foo3)
}

rule OfExample3
{
        strings:
                $foo1 = "foo1"
                $foo2 = "foo2"
                $bar1 = "bar1"
                $bar2 = "bar2"
        condition:
                3 of ($foo*,$bar1,$bar2)
}
```

Вы даже можете использовать ($*) для ссылки на все строки в правиле или написать эквивалентное ключевое слово them для большей наглядности.

```
rule OfExample4
{
        strings:
                $a = "dummy1"
                $b = "dummy2"
                $c = "dummy3"
        condition:
                1 of them // эквивалент для выражения 1 of ($*)
}
```

Во всех приведенных выше примерах число строк задается числовой константой, но может использоваться любое выражение, возвращающее числовое значение. Также могут быть использованы ключевые слова any и all.

```
all of them       // все строки в правиле
any of them       // любая строка в правиле
all of ($a*)      // все строки, начинающиеся с $a
any of ($a,$b,$c) // любая строка из $a, $b или $c
1 of ($*)         // то же самое, что и "any of them"
```

Применение одного и того же условия к нескольким строкам

Есть еще один оператор, который очень похож на оператор of, но более эффективный. Это оператор for...of. Синтаксис данного оператора:

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

И его смысл таков: из строк в string_set по крайней мере expression из них должно удовлетворять условию boolean_expression.

Другими словами: boolean_expression вычисляется для каждой строки из string_set и должно быть хотя бы expression строк, для которых boolean_expression равно True.

Конечно, boolean_expression может быть любым логическим выражением, принятым в разделе condition правила, за исключением одной важной детали: здесь вы можете (и должны) использовать знак доллара ($) в качестве заполнителя для анализируемой строки.

Посмотрите на следующее выражение:

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

Символ $ в булевом выражении не привязан к какой-либо конкретной строке, он будет сначала привязан к строке $a, затем к $b, после чего к $c в трех последовательных вычислениях значения выражения ($ at entrypoint).

Если внимательно посмотреть то видно, что оператор of является частным случаем for...of. Следующие два выражения являются одинаковыми:

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

Можно также использовать символы # и @ для ссылки на число вхождений и первое смещение каждой строки соответственно.

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

Использование анонимных строк с of и for... of

При использовании операторов of и for...of, за которыми следует them, присвоение каждой строке отдельного идентификатора, обычно является лишним. Поскольку мы не ссылаемся на какую-либо строку отдельно, нам не нужно предоставлять уникальный идентификатор для каждой из них. В таких ситуациях можно объявить анонимные строки с идентификаторами, состоящими только из символа $, как в следующем примере:

```
rule AnonymousStrings
{
        strings:
                $ = "dummy1"
                $ = "dummy2"
        condition:
                1 of them
}
```

Перебор строковых вхождений

Как было показано в п. 2.3.2 (Смещение строк или виртуальный адрес), смещения или виртуальные адреса, где строка появляется в адресном пространстве файла или процесса, могут быть доступны с помощью синтаксиса: @a[i], где i - индекс, указывающий, на какое вхождение строки $a вы ссылаетесь.

Иногда необходимо перебирать некоторые из этих смещений и убедиться, что они удовлетворяют заданному условию. Например:

```
rule Occurrences
{
        strings:
                $a = "dummy1"
                $b = "dummy2"
        condition:
                for all i in (1,2,3) : ( @a[i] + 10 == @b[i] )
}
```

Показанное выше правило гласит, что первые три вхождения $b должны быть на расстоянии 10 байт от первых трех вхождений $a.

То же самое условие можно записать и таким образом:

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

Обратите внимание, что мы используем ряд (1..3) вместо перечисления значений индекса (1,2,3). Однако, не обязательно использовать константы для указания границ диапазона, можно также использовать и выражения, как в следующем примере:

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

В этом случае мы перебираем каждое вхождение строки $a (помните, что #a представляет количество вхождений $a). Это правило определяет, что каждое вхождение строки $a должно находиться в пределах первых 100 байт файла.

Если вы хотите выразить, что только некоторые вхождения строки должны удовлетворять условию, то в данном случае применяется та же логика, что и в операторе for...of:

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

Таким образом, синтаксис этого оператора:

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

Ссылки на другие правила

При написании условий для правил можно также ссылаться на ранее определенное правило способом, напоминающим вызов функции в традиционных языках программирования. Таким образом, можно создавать правила, которые зависят от других. Например:

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

        condition:
                $a
}

rule Rule2
{
        strings:
                $a = "dummy2"
```

(continues on next page)

(continued from previous page)

```
    condition:
        $a and Rule1
}
```

Как видно из примера, файл будет удовлетворять правилу Rule2, только если он содержит строку "dummy2" и удовлетворяет правилу Rule1. Обратите внимание, что правило необходимо определить строго до того, как оно будет вызвано.

1.2.4 Еще о правилах

Есть некоторые аспекты правил YARA, которые ранее не были рассмотрены, но очень важны. Это глобальные правила, частные правила, теги и метаданные.

Глобальные правила

Глобальные правила дают вам возможность налагать ограничения во всех ваших правилах сразу. Например, предположим, что вы хотите, чтобы все ваши правила игнорировали те файлы, которые превышают определенный размер. Вы могли бы править все правила, внося необходимые изменения в их условия, или просто написать глобальное правило, подобное этому:

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

Вы можете определить столько глобальных правил, сколько необходимо, они будут проверяться перед остальными правилами, которые, в свою очередь, будут проверяться только в том случае, если все глобальные правила будут выполнены.

Приватные правила

Приватные правила - очень простая концепция. Это правила, которые не сообщают YARA, когда они выполняются при проверке файла. Правила, которые не выдают результат явно, могут показаться на первый взгляд бесполезными, но когда они смешиваются с возможностью ссылаться на одно правило из другого (п. 2.3.11 Ссылки на другие правила)_, они становятся полезными. Приватные правила могут служить блоками для других правил и в то же время предотвращать загромождение вывода YARA нерелевантной информацией.

Чтобы объявить правило как приватное, просто добавьте ключевое слово private перед объявлением правила.

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

Вы можете применить к правилу как модификатор private, так и global, в результате чего о выполнении глобального правила не будет сообщено YARA, но при этом оно будет выполнено.

Тэги правил

Еще одной полезной особенностью YARA является возможность добавления тегов в правила. Эти теги можно использовать позже для фильтрации вывода YARA и показывать вывод только тех правил, которые вас интересуют. В правило можно добавить любое количество тегов, которые объявляются после идентификатора правила, как показано ниже:

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

rule TagsExample2 : Bar
{
...
}
```

Теги должны соответствовать одному и тому же лексическому соглашению для написания идентификаторов правил, поэтому допускаются только буквенно-цифровые символы и подчеркивания, а тег не может начинаться с цифры. Они также чувствительны к регистру.

При использовании YARA вы можете выводить результаты только тех правил, которые помечены тегом или тегами.

Метаданные

Помимо разделов, в которых определены строки и условия, правила могут также иметь раздел метаданных, где можно разместить дополнительную информацию о правиле. Раздел метаданных определяется ключевым словом meta и содержит пары идентификатор/значение, как в следующем примере:

```
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
}
```

Как видно из примера, за идентификаторами метаданных всегда следует знак равенства и присвоенное им значение. Присвоенные значения могут быть строками, числами, или одним из логических значений true или false. Обратите внимание, что пары идентификатор/значение, определенные в разделе метаданные, не могут использоваться в разделе condition, их единственной целью является хранение дополнительной информации о правиле.

1.2.5 Использование модулей

Модули - это расширения базовой функциональности YARA. Некоторые модули, такие как модули PE или Cuckoo, официально распространяются с YARA, а дополнительные могут быть созданы третьими лицами или даже вами самостоятельно, как описано в Главе 4.

Первым шагом к использованию модуля является его импорт с помощью оператора import. Этот оператор должен быть помещен вне любого определения правила и сопровождаться именем модуля, заключенным в двойные кавычки:

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

После импорта модуля вы можете использовать его функции или переменные, используя <имя модуля>. в качестве префикса к любой переменной или функции, экспортируемые модулем. Например:

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

1.2.6 Неопределенные значения

Модули часто оставляют переменные в неопределенном состоянии, например, когда переменная не имеет смысла в текущем контексте (например, pe.entry_point при сканировании файла, отличного от PE-файла). YARA обрабатывает неопределенные значения таким образом, чтобы правило не потеряло свой смысл. Взгляните на это правило:

```
import "pe"

rule Test
{
        strings:
                $a = "some string"

        condition:
                $a and pe.entry_point == 0x1000
}
```

Если сканируемый файл не является PE-файлом, вы не ожидаете, что это правило будет соответствовать файлу, даже если он содержит строку, потому что оба условия (наличие строки и правильное значение для точки входа) должны быть выполнены. Однако, если условие изменено на:

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

В этом случае вы ожидаете, что правило будет соответствовать файлу, если файл содержит строку, даже если это не PE-файл. Именно так ведет себя Яра.

Логика проста: любая арифметическая или логическая операция, а также операция сравнения приведет к неопределенному значению, если один из ее операндов не определен, за исключением операции OR, где неопределенный операнд интерпретируется как false.

1.2.7 Внешние переменные

Внешние переменные позволяют определить правила, которые зависят от значений, предоставляемых извне. Например, можно написать следующее правило:

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

В данном случае ext_var - это внешняя переменная, значение которой присваивается во время выполнения (см. опцию -d командной строки и параметр externals методов compile и match в yara-python). Внешние переменные могут быть целочисленными, строковыми или булевыми, их тип зависит от присвоенного им значения. Целочисленная переменная может заменить любую целочисленную константу в условии, а булевы переменные могут занять место булевых выражений. Например:

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

Внешние переменные строкового типа могут использоваться с операторами: `contains и matches. Оператор contains возвращает true, если строка содержит указанную подстроку. Оператор matches возвращает true, если строка соответствует заданному регулярному выражению.

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

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

Модификаторы регулярных выражений можно использовать вместе с оператором matches, например, если требуется, чтобы регулярное выражение из предыдущего примера не учитывало регистр, можно использовать /[a-z]+/i. Можно также использовать модификатор s для однострочного режима, в этом режиме точка соответствует всем символам, включая разрывы строк. При этом, оба модификатора могут использоваться одновременно, как в следующем примере:

```
rule ExternalVariableExample5
{
        condition:
                /* выбираем однострочный режим без учета регистра */
                string_ext_var matches /[a-z]+/is
}
```

Необходимо иметь в виду, что каждая внешняя переменная, используемая в правилах, должна быть определена во время выполнения либо с помощью опции -d командной строки, либо путем предоставления параметра externals соответствующему методу в yara-python.

1.2.8 Включаемые файлы

Чтобы обеспечить более гибкую организацию файлов правил, YARA предоставляет директиву include. Эта директива работает аналогично директиве препроцессора #include в программах C, которая вставляет содержимое указанного исходного файла в текущий файл во время компиляции. Следующий пример будет включать в себя содержимое файла other.yar в текущий файл:

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

Базовый путь при поиске файла в директиве include будет каталогом, в котором находится текущий файл. По этой причине файл other.yar в предыдущем примере должен находиться в той же директории

текущего файла. Однако, вы также можете указать относительные пути:

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

Или использовать абсолютные пути:

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

В Windows, при указании путей, принимается как прямой, так и обратный слэш, но при этом не забывайте указывать букву диска:

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

1.3 Модули

Модули - это средство, которое YARA предоставляет для расширения своих возможностей. Они позволяют определить структуры данных и функции, которые могут использоваться в правилах для выражения более сложных условий. В этой главе вы найдете описание некоторых модулей, официально распространяемых с YARA, но вы также можете узнать, как писать свои собственные модули в Главе 4 [“Написание собственных модулей”](#ch_4).

1.3.1 Модуль PE

Модуль PE позволяет создавать более детализированные правила для PE-файлов с помощью атрибутов и функций формата PE-файла (детальную информацию о формате PE-файлов можно получить здесь. Этот модуль предоставляет большинство полей, присутствующих в заголовке PE и предоставляет функции, которые могут быть использованы для написания более выразительных и целевых правил. Рассмотрим несколько примеров:

```
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
}
```

Описание

machine

Добавлено в версии 3.3.0.

Целочисленная переменная, содержащая одно из следующих значений:

- MACHINE_UNKNOWN
- MACHINE_AM33
- MACHINE_AMD64
- MACHINE_ARM
- MACHINE_ARMNT
- MACHINE_ARM64
- MACHINE_EBC
- MACHINE_I386
- MACHINE_IA64
- MACHINE_M32R
- MACHINE_MIPS16
- MACHINE_MIPSFPU
- MACHINE_MIPSFPU16
- MACHINE_POWERPC
- MACHINE_POWERPCFP
- MACHINE_R4000
- MACHINE_SH3
- MACHINE_SH3DSP
- MACHINE_SH4
- MACHINE_SH5
- MACHINE_THUMB
- MACHINE_WCEMIPSV2

Пример: pe.machine == pe.MACHINE_AMD64

---

checksum

Добавлено в версии 3.3.0.

Целочисленная переменная, хранящее значение поля CheckSum из OptionalHeader заголовка PE-файла

calculate_checksum

Добавлено в версии 3.3.0.

Функция, рассчитывающая значение контрольной суммы PE-файла

Пример: pe.checksum == pe.calculate_checksum()

---

subsystem

Целочисленная переменная, содержащая одно из следующих значений:

- SUBSYSTEM_UNKNOWN
- SUBSYSTEM_NATIVE
- SUBSYSTEM_WINDOWS_GUI
- SUBSYSTEM_WINDOWS_CUI
- SUBSYSTEM_OS2_CUI
- SUBSYSTEM_POSIX_CUI
- SUBSYSTEM_NATIVE_WINDOWS
- SUBSYSTEM_WINDOWS_CE_GUI
- SUBSYSTEM_EFI_APPLICATION
- SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER
- SUBSYSTEM_EFI_RUNTIME_DRIVER
- SUBSYSTEM_XBOX
- SUBSYSTEM_WINDOWS_BOOT_APPLICATION

Пример: pe.subsystem == pe.SUBSYSTEM_NATIVE

---

timestamp

Переменная, содержащая значение поля TimeDateStamp из FileHeader заголовка PE-файла

---

pointer_to_symbol_table

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_FILE_HEADER::PointerToSymbolTable. Используется, когда PE-образ имеет отладочную информацию COFF.

---

number_of_symbols

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_FILE_HEADER::NumberOfSymbols. Используется, когда PE-образ имеет отладочную информацию COFF.

---

size_of_optional_header

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_FILE_HEADER::SizeOfOptionalHeader. Это реальный размер опционального заголовка (OptionalHeader). Равно 0xE0 для файлов PE32 и 0xF0 для файлов PE32+.

---

opthdr_magic

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::Magic.

---

size_of_code

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfCode. Это сумма размеров необработанных данных в разделах кода.

---

size_of_initialized_data

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfInitializedData.

---

size_of_uninitialized_data

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfUninitializedData.

---

entry_point

Смещение точки входа относительно начала файла или виртуальный адрес в зависимости от того, сканирует ли YARA файл или память процесса соответственно.

---

base_of_code

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::BaseOfCode.

---

base_of_data

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::BaseOfData. Это поле существует только в 32-разрядных PE-файлах.

---

image_base

Базовый адрес загрузки программы.

---

section_alignment

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SectionAlignment. Когда Windows отображает PE-образ в память, все размеры секций (включая размер заголовка) выравниваются до этого значения.

---

file_alignment

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::FileAlignment. Все смещения к данным секции в PE-файле выровнены по этому значению.

---

win32_version_value

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::Win32VersionValue.

---

size_of_image

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfImage. Это общий виртуальный размер заголовка и всех разделов.

---

size_of_headers

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfHeaders. Это размер PE-заголовка PE, включая DOS_заголовок , заголовок файла, опциональный заголовок и все заголовки разделов. Когда PE-файл отображается в память, это значение подлежит выравниванию до SectionAlignment.

---

characteristics

Битовое представление характеристик PE-файла из FileHeader. Каждую характеристику можно проверить, выполнив побитовую операцию AND со следующими константами:

- RELOCS_STRIPPED - Файл не содержит информации о базовых перемещений.
- EXECUTABLE_IMAGE - Файл является исполняемым (т. е. без неразрешенных внешних ссылок).
- LINE_NUMS_STRIPPED - Номера строк были удалены из файла. Этот флаг устарел и должен быть равен нулю.
- LOCAL_SYMS_STRIPPED - Локальные символы, удалены из файла. Этот флаг устарел и должен быть равен нулю.
- AGGRESIVE_WS_TRIM - Принудительное использование файла подкачки. Этот флаг устарел и должен быть равен нулю.
- LARGE_ADDRESS_AWARE - Программа может работать с адресами, большими 2 Гб.
- BYTES_REVERSED_LO - Байты машинного слова меняются местами (little endian). Этот флаг устарел и должен быть равен нулю.
- MACHINE_32BIT - Архитектура 32-разрядного слова.
- DEBUG_STRIPPED - Отладочная информация удалена из PE-файла и вынесена в отдельный .DBG файл.
- REMOVABLE_RUN_FROM_SWAP - Если образ находится на съемном носителе, то его нужно предварительно скопировать в файл подкачки.
- NET_RUN_FROM_SWAP - Если образ находится в сети, то его нужно предварительно скопировать в файл подкачки.

- SYSTEM - Системный файл.
- DLL - DLL-файл.
- UP_SYSTEM_ONLY - Файл должен исполняться только на однопроцессорной машине.
- BYTES_REVERSED_HI - Байты машинного слова меняются местами (big endian). Этот флаг устарел и должен быть равен нулю.

Пример: pe.characteristics & pe.DLL

---

linker_version

Объект с двумя целочисленными атрибутами, по одному для старшей и младшей цифры версии компоновщика.

- major - Старшая цифра версии компоновщика.
- minor - Младшая цифра версии компоновщика.

---

os_version

Объект с двумя целочисленными атрибутами, по одному для старшей и младшей цифры версии операционной системы.

- major - Старшая цифра версии операционной системы.
- minor - Младшая цифра версии операционной системы.

---

image_version

Объект с двумя целочисленными атрибутами, по одному для старшей и младшей цифры версии файла.

- major - Старшая цифра версии файла.
- minor - Младшая цифра версии файла.

---

subsystem_version

Объект с двумя целочисленными атрибутами, по одному для старшей и младшей цифры версии подсистемы.

- major - Старшая цифра версии подсистемы.
- minor - Младшая цифра версии подсистемы.

---

dll_characteristics

Битовое представление дополнительных характеристик PE-файла из OptionalHeader. Не путайте эти характеристики с характеристиками из FileHeader. Каждую характеристику можно проверить, выполнив побитовую операцию AND со следующими константами:

- DYNAMIC_BASE - Файл может быть перемещен (файл совместимый с ASLR).
- FORCE_INTEGRITY
- NX_COMPAT - Файл совместимый с DEP.
- NO_ISOLATION

- NO_SEH - Файл не содержит структурированных обработчиков исключений, он должен быть настроен на использование SafeSEH.
- NO_BIND
- WDM_DRIVER - Файл является WDM-драйвером.
- TERMINAL_SERVER_AWARE - файл совместимый с сервером терминалов.

---

size_of_stack_reserve

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfStackReserve. Это объем виртуальной памяти по умолчанию, который будет зарезервирован для стека.

---

size_of_stack_commit

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfStackCommit. Это объем виртуальной памяти по умолчанию, который будет выделен для стека.

---

size_of_heap_reserve

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfHeapReserve. Это объем виртуальной памяти по умолчанию, который будет зарезервирован для кучи основного процесса.

---

size_of_heap_commit

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::SizeOfHeapCommit. Это объем виртуальной памяти по умолчанию, который будет выделен для кучи основного процесса.

---

loader_flags

Добавлено в версии 3.8.0.

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::LoaderFlags.

---

number_of_rva_and_sizes

Переменная, содержащая значение IMAGE_OPTIONAL_HEADER::NumberOfRvaAndSizes. Это число элементов в массиве IMAGE_OPTIONAL_HEADER::DataDirectory.

---

data_directories

Добавлено в версии 3.8.0.

Массив каталогов данных. Каждый каталог данных содержит виртуальный адрес и длину соответствующего каталога данных. Каждый каталог данных содержит следующие записи:

- virtual_address - Относительный виртуальный адрес (RVA) каталога данных. Если это ноль, то каталог данных отсутствует.
- size - Размер каталога данных в байтах.

Индекс записи каталога данных может иметь одно из следующих значений:

- IMAGE_DIRECTORY_ENTRY_EXPORT - Каталог для экспортируемых функций.
- IMAGE_DIRECTORY_ENTRY_IMPORT - Каталог для импортируемых функций.
- IMAGE_DIRECTORY_ENTRY_RESOURCE - Каталог для ресурсов.
- IMAGE_DIRECTORY_ENTRY_EXCEPTION - Каталог информации об исключениях.
- IMAGE_DIRECTORY_ENTRY_SECURITY - Указатель на таблицу сертификатов цифровых подписей. Если цифровая подпись отсутствует, будет содержать нули.
- IMAGE_DIRECTORY_ENTRY_BASERELOC - Каталог таблицы переадресации.
- IMAGE_DIRECTORY_ENTRY_DEBUG - Каталог для отладочной информации.
- IMAGE_DIRECTORY_ENTRY_TLS - Каталог TLS (локальной памяти потоков).
- IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG - Каталог конфигурации загрузки.
- IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT - Каталог для таблицы диапазонного импорта.
- IMAGE_DIRECTORY_ENTRY_IAT - Каталог для таблицы адресов импорта (IAT).
- IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT - Каталог для таблицы отложенного импорта. Структура таблицы отложенного импорта зависит от компоновщика.
- IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR - Каталог для заголовков .NET.

Пример: pe.data_directories[pe.IMAGE_DIRECTORY_ENTRY_EXPORT].virtual_address != 0

---

number_of_sections

Число секций в PE-файле.

---

sections

Добавлено в версии 3.3.0.

Начинающийся с нуля массив объектов описания секций, по одному на каждую секцию, которые имеет PE-файл. Доступ к отдельным объектам массива можно получить с помощью оператора []. Каждый объект массива имеет следующие атрибуты:

- name - Имя секции.
- characteristics - Характеристики секции.
- virtual_address - Виртуальный адрес секции.
- virtual_size - Виртуальный размер секции.
- raw_data_offset - Смещение секции в файле.
- raw_data_size - Физический размер секции.
- pointer_to_relocations - Добавлено в версии 3.8.0. Переменная, содержащая значение IMAGE_SECTION_HEADER::PointerToRelocations.

- pointer_to_line_numbers - Добавлено в версии 3.8.0. Переменная, содержащая значение IMAGE_SECTION_HEADER::PointerToLinenumbers.
- number_of_relocations Добавлено в версии 3.8.0. Переменная, содержащая значение IMAGE_SECTION_HEADER::NumberOfRelocations.
- number_of_line_numbers - Добавлено в версии 3.8.0. Переменная, содержащая значение IMAGE_SECTION_HEADER::NumberOfLineNumbers.

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

Каждую характеристику секции можно проверить, выполнив побитовую операцию AND со следующими константами:

- SECTION_CNT_CODE
- SECTION_CNT_INITIALIZED_DATA
- SECTION_CNT_UNINITIALIZED_DATA
- SECTION_GPREL
- SECTION_MEM_16BIT
- SECTION_LNK_NRELOC_OVFL
- SECTION_MEM_DISCARDABLE
- SECTION_MEM_NOT_CACHED
- SECTION_MEM_NOT_PAGED
- SECTION_MEM_SHARED
- SECTION_MEM_EXECUTE
- SECTION_MEM_READ
- SECTION_MEM_WRITE

Пример: pe.sections[1].characteristics & SECTION_CNT_CODE

---

overlay

Добавлено в версии 3.6.0.

Структура, содержащая следующие целочисленные элементы:

- offset - Смещение секции оверлея.
- size - размер секции оверлея.

Пример: uint8(0x0d) at pe.overlay.offset and pe.overlay.size > 1024

---

number_of_resources

Число ресурсов в PE-файле

---

resource_timestamp

Дата и время подключения ресурсов от ресурсного компилятора. Сохраняется в виде целого числа.

---

resource_version

Объект, содержащий два целых числа:

- major - Старшая цифра номера версии ресурсов.
- minor - Младшая цифра номера версии ресурсов.

---

resources

Добавлено в версии 3.3.0.

Начинающийся с нуля массив объектов описания ресурсов, по одному на каждый ресурс, который имеет PE-файл. Доступ к отдельным объектам массива можно получить с помощью оператора []. Каждый объект массива имеет следующие атрибуты:

- offset - Смещение на ресурс.
- length - Длина ресурса.
- type - Тип ресурса (integer).
- id - Идентификатор ресурса (integer).
- language - Язык ресурса (integer).
- type_string - Тип ресурса в виде строки, если указан.
- name_string - Имя ресурса в виде строки, если указан.
- language_string - Язык ресурса в виде строки, если указан.

Все ресурсы должны иметь определенный тип, идентификатор (имя) и язык. Они могут выражены либо целыми числами, либо в виде строк.

Пример: pe.resources[0].type == pe.RESOURCE_TYPE_RCDATA

Пример: pe.resources[0].name_string == "F\x00I\x00L\x00E\x00"

Типы ресурсов можно проверить с помощью следующих констант:

- RESOURCE_TYPE_CURSOR
- RESOURCE_TYPE_BITMAP
- RESOURCE_TYPE_ICON
- RESOURCE_TYPE_MENU
- RESOURCE_TYPE_DIALOG
- RESOURCE_TYPE_STRING
- RESOURCE_TYPE_FONTDIR
- RESOURCE_TYPE_FONT
- RESOURCE_TYPE_ACCELERATOR
- RESOURCE_TYPE_RCDATA
- RESOURCE_TYPE_MESSAGETABLE
- RESOURCE_TYPE_GROUP_CURSOR
- RESOURCE_TYPE_GROUP_ICON
- RESOURCE_TYPE_VERSION

- RESOURCE_TYPE_DLGINCLUDE
- RESOURCE_TYPE_PLUGPLAY
- RESOURCE_TYPE_VXD
- RESOURCE_TYPE_ANICURSOR
- RESOURCE_TYPE_ANIICON
- RESOURCE_TYPE_HTML
- RESOURCE_TYPE_MANIFEST

Для получения дополнительной информации см.:

http://msdn.microsoft.com/en-us/library/ms648009(v=vs.85).aspx

---

version_info

Добавлено в версии 3.2.0.

Словарь, содержащий информацию о версии PE-файла. Типичные ключи:

Comments, CompanyName, FileDescription, FileVersion, InternalName, LegalCopyright, LegalTrademarks, OriginalFilename, ProductName, ProductVersion

Для получения дополнительной информации см.:

http://msdn.microsoft.com/en-us/library/windows/desktop/ms646987(v=vs.85).aspx

Пример: pe.version_info[“CompanyName”] contains "Microsoft"

---

number_of_signatures

Число authenticode-подписей в PE-файле.

---

signatures

Начинающийся с нуля массив объектов описания подписи, по одному для каждой authenticode-подписи в PE-файле. Обычно PE-файлы имеют одну подпись.

- thumbprint - Добавлено в версии 3.8.0. Строка, содержащая отпечаток (криптографический хэш) подписи.
- issuer - Строка, содержащая информацию об эмитенте подписи.

Вот несколько примеров:

```
"/C=US/ST=Washington/L=Redmond/O=Microsoft Corporation/CN=Microsoft Code Signing PCA"
"/C=US/O=VeriSign, Inc./OU=VeriSign Trust Network/OU=Terms of use at https://www.verisign.com/rpa
(c)10/CN=VeriSign Class 3 Code Signing 2010 CA"
"/C=GB/ST=Greater Manchester/L=Salford/O=COMODO CA Limited/CN=COMODO Code Signing CA 2"
```

- subject - Строка, содержащая информацию о субъекте.
- version - Номер версии.
- algorithm - Алгоритм, используемый в подписи. Обычно “sha1WithRSAEncryption”.
- serial - Строка, содержащая серийный номер.

Например:

```
"52:00:e5:aa:25:56:fc:1a:86:ed:96:c9:d4:4b:33:c7"
```

- not_before - Временная метка в формате Unix, с которой начинается срок действия этой подписи.
- not_after - Временная метка в формате Unix, на которой заканчивается срок действия этой подписи.
- valid_on(timestamp) - Функция возвращает true, если подпись действительна на дату, указанную меткой времени timestamp.

Например, выражение:

```
pe.signatures[n].valid_on(timestamp)
```

эквивалентно следующему выражению:

```
timestamp >= pe.signatures[n].not_before and timestamp <= pe.signatures[n].not_after
```

---

rich_signature

Структура, содержащая информацию о Rich-сигнатуре PE-файла. Подробное описание Rich-сигнатуры можно найти здесь.

- offset - Смещение начала Rich-сигнатуры. Будет не определено если Rich-сигнатура отсутствует.
- length - Длина Rich-сигнатуры, не включающая конечный маркер сигнатуры “Rich”.
- key - Ключ, для расшифровки данных с помощью XOR.
- raw_data - Необработанные данные, как они отображаются в файле.
- clear_data - Данные после расшифровки.
- version(version, [toolid]) - Добавлено в версию 3.5.0. Функция, возвращающая true, если PE-файл имеет указанную версию version в Rich-сигнатуре. Укажите необязательный аргумент toolid для сопоставления только в том случае, если оба аргумента совпадают для одной записи. Более подробную информацию можно найти здесь.

Пример: pe.rich_signature.version(21005)

- toolid(toolid, [version]) - Добавлено в версии 3.5.0. Функция, возвращающая true, если PE-файл имеет указанный идентификатор toolid в Rich-сигнатуре. Укажите необязательный аргумент toolid для сопоставления только в том случае, если оба аргумента совпадают для одной записи. Более подробную информацию можно найти здесь.

Пример: pe.rich_signature.toolid(222)

---

exports(function_name)

Функция, возвращающая true, если PE-файл экспортирует функцию function_name или false в противном случае.

Пример: pe.exports("CPlApplet")

---

exports(ordinal)

Добавлено в версии 3.6.0.

Функция, возвращающая true, если PE-файл экспортирует функцию по ординалу ordinal или false в противном случае.

Пример: pe.exports(72)

---

exports(/regular_expression/)

Добавлена в версии 3.7.1.

Функция, возвращающая true, если PE-файл экспортирует функции в соответствии с регулярным выражением /regular_expression/ или false в противном случае.

Пример: pe.exports(/^AXS@@/)

---

number_of_exports

Добавлено в версии 3.6.0.

Число экспортов в PE-файле.

---

number_of_imports

Добавлено в версии 3.6.0.

Число импортов в PE-файле.

---

imports(dll_name, function_name)

Функция, возвращающая true, если PE-файл импортирует функцию function_name из библиотеки dll_name, или false в противном случае (dll_name не чувствительна к регистру).

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

---

imports(dll_name)

Добавлено в версии 3.5.0.

Функция, возвращающая true, если PE-файл импортирует что-либо из библиотеки dll_name, или false в противном случае (dll_name не чувствительна к регистру).

Пример: pe.imports("kernel32.dll")

---

imports(dll_name, ordinal)

Добавлено в версии 3.5.0.

Функция, возвращающая true, если PE-файл импортирует функцию по ординалу ordinal из библиотеки dll_name, или false в противном случае (dll_name не чувствительна к регистру).

Пример: pe.imports("WS2_32.DLL", 3)

---

imports(dll_regexp, function_regexp)

Добавлено в версии 3.8.0.

Функция, возвращающая true, если PE-файл импортирует функции в соответствии с регулярным выражением function_regexp из библиотеки в соответствии с регулярным выражением dll_regexp или false в противном случае. dll_regexp чувствителен к регистру, если не используется модификатор /i в регулярном выражении, как показано ниже.

Пример: pe.imports(/kernel32.dll/i, /(Read|Write)ProcessMemory/)

---

locale(locale_identifier)

Добавлено в версии 3.2.0.

Функция, возвращающая true, если PE-файл имеет ресурс с указанным идентификатором локали locale_identifier. Идентификаторы локали являются 16-разрядными целыми числами и могут быть найдены здесь.

Пример: pe.locale(0x0419) // Россия (RU)

---

language(language_identifier)

Добавлено в версии 3.2.0.

Функция, возвращающая true, если PE-файл имеет ресурс с указанным идентификатором языка language_identifier. Идентификаторы языка представляют собой 8-разрядные целые числа и могут быть найдены здесь.

Пример: pe.language(0x0A) // Испания

---

imphash()

Добавлено в версии 3.2.0.

Функция, возвращающая хэш импорта или imphash для PE-файла. Imphash - это MD5-хэш таблицы импорта PE-файла после некоторой нормализации. Imphash для PE-файла может быть также вычислена с помощью pefile и вы можете найти больше информации в Mandiant's blog.

Пример: pe.imphash() == "b8bb385806b89680e13fc0cf24f4431e"

---

section_index(name)

Функция, возвращающая индекс секции с именем name (name чувствительно к регистру).

Пример: pe.section_index(".TEXT")

---

section_index(addr)

Добавлено в версии 3.3.0.

Функция, возвращающая индекс секции с адресом addr. Адрес addr может быть смещением в файле или адресом в памяти.

Пример: pe.section_index(pe.entry_point)

---

is_dll()

Добавлено в версии 3.5.0.

Функция возвращает true если PE-файл является DLL-библиотекой.

Пример: pe.is_dll()

---

is_32bit()

Добавлено в версии 3.5.0.

Функция возвращает true если PE-файл является 32-битным.

Пример: pe.is_32bit()

---

is_64bit()

Добавлено в версии 3.5.0.

Функция возвращает true если PE-файл является 64-битным.

Пример: pe.is_64bit()

---

rva_to_offset(addr)

Добавлено в версии 3.6.0.

Функция, возвращающая смещение в файле для RVA-адреса addr.

Пример: pe.rva_to_offset(pe.entry_point)

1.3.2 Модуль ELF

Добавлено в версии 3.2.0.

Модуль ELF очень похож на модуль PE, но предназначен для анализа файлов типа ELF. Этот модуль предоставляет большинство полей, присутствующих в заголовке ELF-файлов. Рассмотрим несколько примеров:

```
import "elf"
rule single_section
{
        condition:
                elf.number_of_sections == 1
}

rule elf_64
{
        condition:
                elf.machine == elf.EM_X86_64
}
```

Описание

type

Целочисленная переменная с одним из следующих значений:

- ET_NONE - Тип файла не определен.
- ET_REL - Перемещаемый файл.
- ET_EXEC - Исполняемый файл.
- ET_DYN - Общий объектный файл.
- ET_CORE - Файл ядра.

Пример: elf.type == elf.ET_EXEC

---

machine

Целочисленная переменная с одним из следующих значений:

- EM_M32
- EM_SPARC
- EM_386
- EM_68K
- EM_88K
- EM_860
- EM_MIPS
- EM_MIPS_RS3_LE
- EM_PPC
- EM_PPC64
- EM_ARM
- EM_X86_64
- EM_AARCH64

Пример: elf.machine == elf.EM_X86_64

---

entry_point

Смещение точки входа в файле или виртуальный адрес в зависимости от того, сканирует ли YARA файл или память процесса соответственно.

---

number_of_sections

Число секций в ELF-файле.

---

sections

Начинающийся с нуля массив объектов описания секций, по одному на каждую секцию, которые имеет ELF-файл. Доступ к отдельным объектам массива можно получить с помощью оператора []. Каждый объект массива имеет следующие атрибуты:

- name - Имя секции.

Пример: elf.sections[3].name == ".bss"

- size - Размер секции в байтах. За исключением секций типа SHT_NOBITS (см. тип type секции ниже), секция занимает sh_size байт в файле. Раздел SHT_NOBITS может иметь ненулевой размер в памяти, но он не занимает места в файле.
- offset - Смещение от начала файла до первого байта секции. Один из типов секции SHT_NOBITS, который будет описан ниже, не занимает места в файле, а его элемент offset определяет абстрактное размещение в файле.
- type - Целочисленная переменная с одним из следующих значений:
  - SHT_NULL - Этим значением отмечены неактивные секции. Остальные поля описаний таких секций имеют неопределенное значение.
  - SHT_PROGBITS - Раздел содержит информацию, формат и значение которой определяются исключительно программой (код, данные или что-либо еще).
  - SHT_SYMTAB - Секция содержит таблицу символов.
  - SHT_STRTAB - Секция содержит таблицу строк. Объектный файл может иметь несколько секций с таблицами строк.
  - SHT_RELA - Секция содержит записи о перемещаемых адресах (relocations).
  - SHT_HASH - Секция содержит хеш-таблицу имен для динамического связывания.
  - SHT_DYNAMIC - Секция содержит информацию для динамического связывания.
  - SHT_NOTE - Секция содержит дополнительную информацию.
  - SHT_NOBITS - Секция этого типа не занимает места в файле, но в остальном напоминает секцию типа SHT_PROGBITS.
  - SHT_REL - Секция содержит записи о перемещаемых адресах.
  - SHT_SHLIB - Этот тип секции зарезервирован.
  - SHT_DYNSYM - Секция содержит набор символов для динамической компоновки.
- flags - Целочисленная переменная, в которой содержатся флаги секции, определяемые следующим образом:
  - SHF_WRITE - Секция содержит данные, которые должны быть доступны для записи во время выполнения процесса.
  - SHF_ALLOC - Секция занимает память при работе процесса. Некоторые управляющие секции не располагаются в образе памяти объектного файла. Этот атрибут выключен у таких разделов.
  - SHF_EXECINSTR - Секция содержит исполняемые машинные инструкции.

Пример: elf.sections[2].flags & elf.SHF_WRITE

- address - Добавлено в версии 3.6.0. Виртуальный адрес, с которого начинается секция.

---

number_of_segments

Добавлено в версии 3.4.0.

Число сегментов в ELF-файле.

---

segments

Добавлено в версии 3.4.0.

Начинающийся с нуля массив объектов описания сегментов, по одному на каждый сегмент, которые имеет ELF-файл. Доступ к отдельным объектам массива можно получить с помощью оператора []. Каждый объект массива имеет следующие атрибуты:

- alignment - Значение согласно которому сегменты выровнены в памяти и в файле.
- file_size - Число байт занимаемое сегментом в файле. Оно может быть равно нулю.
- flags - Комбинация флагов сегмента:
  - PF_R - Сегмент доступен для чтения.
  - PF_W - Сегмент доступен для записи.
  - PF_X - Исполняемый сегмент.
- memory_size - Размер сегмента в памяти.
- offset - Это поле содержит смещение от начала файла, по которому располагается первый байт сегмента.
- physical_address - В системах, для которых важна физическая адресация, это поле содержит физический адрес сегмента.
- type - Тип сегмента, определяемый одним из следующих значений:
  - PT_NULL
  - PT_LOAD
  - PT_DYNAMIC
  - PT_INTERP
  - PT_NOTE
  - PT_SHLIB
  - PT_PHDR
  - PT_LOPROC
  - PT_HIPROC
  - PT_GNU_STACK
- virtual_address - Это поле содержит виртуальный адрес, по которому располагается первый байт сегмента в памяти.

---

dynamic_section_entries

Добавлено в версии 3.6.0.

Число записей в секции .dynamic ELF-файла

---

dynamic

Добавлено в версии 3.6.0.

Начинающийся с нуля массив объектов, по одному на каждую запись секции .dynamic ELF-файла. Доступ к отдельным объектам массива можно получить с помощью оператора []. Каждый объект массива имеет следующие атрибуты:

- type - Значение, которое описывает тип секции .dynamic. Возможные значения:
  - DT_NULL
  - DT_NEEDED
  - DT_PLTRELSZ
  - DT_PLTGOT
  - DT_HASH
  - DT_STRTAB
  - DT_SYMTAB
  - DT_RELA
  - DT_RELASZ
  - DT_RELAENT
  - DT_STRSZ
  - DT_SYMENT
  - DT_INIT
  - DT_FINI
  - DT_SONAME
  - DT_RPATH
  - DT_SYMBOLIC
  - DT_REL
  - DT_RELSZ
  - DT_RELENT
  - DT_PLTREL
  - DT_DEBUG
  - DT_TEXTREL
  - DT_JMPREL
  - DT_BIND_NOW
  - DT_INIT_ARRAY
  - DT_FINI_ARRAY
  - DT_INIT_ARRAYSZ
  - DT_FINI_ARRAYSZ
  - DT_RUNPATH
  - DT_FLAGS
  - DT_ENCODING

- value - Значение, связанное с данным типом. Тип значения (адрес, размер и т. д.) зависит от типа записи.

---

symtab_entries

Добавлено в версии 3.6.0.

Число записей в таблице символов в ELF-файле.

---

symtab

Добавлено в версии 3.6.0.

Начинающийся с нуля массив описаний символьных объектов, по одному на каждую запись, найденную в SYMBTAB ELF-файла. Доступ к отдельным символьным объектам можно получить с помощью оператора []. Каждый символьный объект имеет следующие атрибуты:

- name - Имя символа.
- value - Значение, связанное с символом. Обычно, виртуальный адрес.
- size - Размер символа.
- type - Тип символа. Возможные значения:
  - STT_NOTYPE
  - STT_OBJECT
  - STT_FUNC
  - STT_SECTION
  - STT_FILE
  - STT_COMMON
  - STT_TLS
- bind - Атрибуты привязки символа. Возможные значения:
  - STB_LOCAL
  - STB_GLOBAL
  - STB_WEAK
- shndx - Индекс секции, с которым связан символ.

1.3.3 Модуль Cuckoo

Модуль Cuckoo позволяет создавать правила YARA на основе поведенческой информации, генерируемой Cuckoo sandbox. При сканировании PE-файла с помощью YARA вы можете передать дополнительную информацию о его поведении модулю cuckoo и создавать правила, основанные не только на том, что содержит файл, но и на том, что он делает.

> Warning: Этот модуль не встроен в YARA по умолчанию, чтобы узнать, как его включить, обратитесь к п. 1.1.

Для пользователей Windows: этот модуль уже включен в официальные бинарные файлы Windows.

Предположим, что вы заинтересованы в том, чтобы исполняемые файлы отправляли HTTP-запросы на http://someone.doingevil.com. В предыдущих версиях YARA вам приходилось довольствоваться только этим:

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

Проблема с этим правилом заключается в том, что доменное имя может содержаться в файле по вполне обоснованным причинам, не связанным с отправкой HTTP-запросов на http://someone.doingevil.com. Кроме того, вредоносный файл может содержать имя домена в зашифрованном или обфусцированном виде, в этом случае это правило будет полностью бесполезным.

Но теперь с модулем Cuckoo вы можете взять отчет о поведении, сгенерированный для исполняемого файла вашей песочницей Cuckoo, передать его вместе с исполняемым файлом в YARA и написать правило, подобное этому:

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

Конечно, вы можете смешать ваши связанные с поведением условия с обычными условиями на основе строк:

```
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/)
}
```

Но как мы можем передать информацию о поведении модулю Cuckoo? В случае использования командной строки необходимо использовать опцию -x следующим образом:

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

behavior_report_file - это путь к файлу, содержащему файл поведения, сгенерированный песочницей Cuckoo в формате JSON.

Если вы используете yara-python, вы должны передать отчет о поведении в аргументе modules_data для метода match:

```
import yara
rules = yara.compile('./rules_file')
```

(continues on next page)

(continued from previous page)

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

Описание

network

- http_request(regexp) - Функция возвращает true, если программа отправила HTTP-запрос на URL-адрес, соответствующий регулярному выражению regexp.

Пример: cuckoo.network.http_request(/evil.com/)

- http_get(regexp) - Аналогичен http_request(), но учитывает только запросы GET.
- http_post(regexp) - Аналогичен http_request (), но учитывает только запросы POST.
- dns_lookup(regexp) - Функция возвращает true, если программа отправила запрос на разрешение имени домена, соответствующего указанному регулярному выражению.

Пример: cuckoo.network.dns_lookup(/evil.com/)

---

registry

- key_access(regexp) - Функция возвращает true, если программа произвела обращение к записи реестра, соответствующей регулярному выражению regexp.

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

---

filesystem

- file_access(regexp) - Функция возвращает true, если программа произвела обращение к файлу, соответствующему регулярному выражению regexp.

Пример: cuckoo.filesystem.file_access(/autoexec.bat/)

---

sync

- mutex(regexp) - Функция возвращает true, если программа открыла и создала мьютекс, соответствующий регулярному выражению regexp.

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

1.3.4 Модуль Magic

Добавлено в версии 3.1.0.

Модуль Magic позволяет определить тип файла, на основе вывода стандартной команды Unix - file.

> Warning: Этот модуль не встроен в YARA по умолчанию, чтобы узнать, как его включить, обратитесь к п. 1.1.

Note: Для пользователей Windows: данный модуль не поддерживается Windows.

В этом модуле есть две функции: type () и mime_type (). Первая возвращает описательную строку, возвращаемую командой file, например, если вы запустите file для какого-либо документа PDF, вы получите что-то вроде этого:

```
$file some.pdf
some.pdf: PDF document, version 1.5
```

Функция type () в этом случае возвращает "PDF document, version 1.5". Использование функции mime_type () аналогично передаче аргумента --mime для команды file:

```
$file --mime some.pdf
some.pdf: application/pdf; charset=binary
```

mime_type () вернет "application/pdf" без части charset.

Немного поэкспериментировав с командой file, вы можете узнать, какие выходные данные ожидать для разных типов файлов. Вот несколько примеров:

- JPEG image data, JFIF standard 1.01
- PE32 executable for MS Windows (GUI) Intel 80386 32-bit
- PNG image data, 1240 x 1753, 8-bit/color RGBA, non-interlaced
- ASCII text, with no line terminators
- Zip archive data, at least v2.0 to extract

type()

Функция, возвращающая строку с типом файла.

Пример: magic.type() contains "PDF"

mime_type()

Функция, возвращающая строку с типом MIME файла.

Пример: magic.mime_type() == "application/pdf"

1.3.5 Модуль Hash

Добавлено в версии 3.2.0.

Модуль Hash позволяет вычислять хэши (MD5, SHA1, SHA256) из частей файла и создавать сигнатуры на основе этих хэшей.

Warning: Этот модуль зависит от библиотеки OpenSSL. Пожалуйста, обратитесь к п. 1.1 для получения информации о том, как встроить OpenSSL-зависимые функции в YARA.

Note: Для пользователей Windows: этот модуль уже включен в официальные бинарные файлы.

---

md5(offset, size)

Возвращает MD5-хэш для size байтов, начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен быть виртуальным адресом в адресном пространстве процесса. Возвращаемая строка всегда в нижнем регистре.

Пример: hash.md5(0, filesize) == "feba6c919e3797e7778e8f2e85fa033d"

---

md5(string)

Возвращает MD5-хэш строки string.

Example: hash.md5(“dummy”) == "275876e34cf609db118f3d84b799a790"

---

sha1(offset, size)

Возвращает SHA1-хэш для size байтов, начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен быть виртуальным адресом в адресном пространстве процесса. Возвращаемая строка всегда в нижнем регистре..

---

sha1(string)

Возвращает SHA1-хэш строки string.

---

sha256(offset, size)

Возвращает SHA256-хэш для size байтов, начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен быть виртуальным адресом в адресном пространстве процесса. Возвращаемая строка всегда в нижнем регистре..

---

sha256(string)

Возвращает SHA256-хэш строки string.

---

checksum32(offset, size)

Возвращает 32-разрядную контрольную сумму для size байтов, начиная со смещения offset. Контрольная сумма - это сумма всех байтов (без знака).

---

checksum32(string)

Возвращает 32-разрядную контрольную сумму строки string. Контрольная сумма - это сумма всех байтов (без знака).

1.3.6 Модуль Math

Добавлено в версии 3.3.0.

Модуль Math позволяет вам вычислять определенные значения из частей вашего файла и создавать сигнатуры на основе этих результатов.

Note: Где отмечено, функции модуля возвращают числа с плавающей запятой. YARA может преобразовывать целые числа в числа с плавающей запятой во время большинства операций. Пример, приведенный ниже автоматически преобразует 7 в 7.0, потому что тип возвращаемой функции энтропии - значение с плавающей запятой:

```
math.entropy(0, filesize) >= 7
```

Единственным исключением является случай, когда функции требуется число с плавающей запятой в качестве аргумента. Например, такая запись приведет к синтаксической ошибке, поскольку аргументы должны быть числами с плавающей запятой:

```
math.in_range(2, 1, 3)
```

---

entropy(offset, size)

Возвращает энтропию size байт начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен содержать виртуальный адрес в адресном пространстве процесса. Возвращаемое значение - число с плавающей запятой.

Пример: math.entropy(0, filesize) >= 7

---

entropy(string)

Возвращает энтропию строки string.

Пример: math.entropy("dummy") > 7

---

monte_carlo_pi(offset, size)

Возвращает процент от числа Pi при расчете числа Pi методом Монте-Карло с использованием последовательности чисел размером size байт, начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен содержать виртуальный адрес в адресном пространстве процесса. Возвращаемое значение - число с плавающей запятой.

Пример: math.monte_carlo_pi(0, filesize) < 0.0

---

monte_carlo_pi(string)

Возвращает процент от числа Pi при расчете числа Pi методом Монте-Карло с использованием строки string.

---

serial_correlation(offset, size)

Возвращает значение коэффициента последовательной корреляции для size байт, начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен содержать виртуальный адрес в адресном пространстве процесса. Возвращаемое значение - число с плавающей запятой в пределах от 0.0 до 1.0.

Пример: math.serial_correlation(0, filesize) < 0.2

---

serial_correlation(string)

Возвращает значение коэффициента последовательной корреляции для строки string.

---

mean(offset, size)

Возвращает среднее значение для size байт, начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен содержать виртуальный адрес в адресном пространстве процесса. Возвращаемое значение - число с плавающей запятой.

Пример: math.mean(0, filesize) < 72.0

---

mean(string)

Возвращает среднее значение для строки string.

---

deviation(offset, size, mean)

Возвращает отклонение от среднего значения для size байт, начиная со смещения offset. При сканировании запущенного процесса аргумент offset должен содержать виртуальный адрес в адресном пространстве процесса. Возвращаемое значение - число с плавающей запятой.

Среднее значение равномерно распределенной случайной выборки байтов равно числу 127.5, которое доступно как константа math.MEAN_BYTES.

Пример: math.deviation(0, filesize, math.MEAN_BYTES) == 64.0

---

deviation(string, mean)

Возвращает отклонение от среднего значения для строки string.

---

in_range(test, lower, upper)

Возвращает true, если значение test находится между нижним lower и верхним upper значениями. Сравнение производится включительно для lower и upper значений.

Пример: math.in_range(math.deviation(0, filesize, math.MEAN_BYTES), 63.9, 64,1)

---

max(int, int)

Добавлено в версии 3.8.0.

Возвращает максимум из двух целочисленных беззнаковых значений.

---

min(int, int)

Добавлено в версии 3.8.0.

Возвращает минимум из двух целочисленных беззнаковых значений.

1.3.7 Модуль dotnet

Добавлено в версии 3.6.0.

Модуль dotnet позволяет создавать более детализированные правила для файлов .NET с помощью атрибутов и функций формата файлов .NET. Например:

```
import "dotnet"
rule not_exactly_five_streams
{
        condition:
                dotnet.number_of_streams != 5
}

rule blop_stream
{
        condition:
                for any i in (0..dotnet.number_of_streams - 1):
                        (dotnet.streams[i].name == "#Blop")
}
```

Описание

version

Строка с версией, содержащаяся в корне метаданных.

Пример: dotnet.version == "v2.0.50727"

---

module_name

Наименование модуля.

Example: dotnet.module_name == "axs"

---

number_of_streams

Число потоков в файле.

---

streams

Начинающийся с нуля массив объектов описания потоков, для каждого потока в файле. Доступ к отдельным объектам массива можно получить с помощью оператора []. Каждый объект массива имеет следующие атрибуты:

- name - Имя потока.
- offset - Смещение потока.
- size - Размер потока.

Пример: dotnet.streams[0].name == "#~"

---

number_of_guids

Количество идентификаторов в GUID-массиве.

---

guids

Начинающийся с нуля массив строк, по одной для каждого GUID. Доступ к отдельным объектам массива можно получить с помощью оператора [].

Пример: dotnet.guids[0] == "99c08ffd-f378-a891-10ab-c02fe11be6ef"

---

number_of_resources

Число ресурсов в .NET-файле. Они отличаются от обычных ресурсов PE-файлов.

resources

Начинающийся с нуля массив объектов описания ресурсов, для каждого ресурса в файле. Доступ к отдельным объектам массива можно получить с помощью оператора []. Каждый объект массива имеет следующие атрибуты:

- offset - Смещение на данные ресурса.
- length - Длина данных ресурса.
- name - Имя ресурса (в виде строки).

Пример: uint16be(dotnet.resources[0].offset) == 0x4d5a

---

assembly

Объект, содержащий информацию о сборке .NET:

- version - Объект с целочисленными значениями, представляющими информацию о версии для этой сборки. Атрибуты: major minor build_number revision_number
- name - Строка, содержащая имя сборки.
- culture - Строка, содержащая language/country/region данной сборки.

Пример: dotnet.assembly.name == "Keylogger"

Пример: dotnet.assembly.version.major == 7 and dotnet.assembly.version.minor == 0

---

number_of_modulerefs

Число ссылок на модули в .NET-файле.

---

modulerefs

Начинающийся с нуля массив строк, по одной на каждую ссылку на модуль в .NET-файле. Доступ к отдельным объектам массива можно получить с помощью оператора [].

Example: dotnet.modulerefs[0] == "kernel32"

---

typelib

Библиотека типа .NET-файла.

---

assembly_refs

Объект для справочной информации сборки .NET.

- version -Объект с целочисленными значениями, представляющими информацию о версии для этой сборки. Атрибуты: major minor build_number revision_number.
- name - Строка, содержащая имя сборки.
- public_key_or_token - Строка, содержащая открытый ключ или токен, который идентифицирует автора этой сборки.

---

number_of_user_strings

Число пользовательских строк в .NET-файле.

---

user_strings

Начинающийся с нуля массив пользовательских строк, по одной на каждый поток, содержащийся в .NET-файле. Доступ к отдельным строкам можно получить с помощью оператора [].

1.3.8 Модуль Time

Добавлено в версии 3.7.0.

Модуль Time позволяет использовать временные условия в правилах YARA.

now()

Функция возвращает целое число - количество секунд с 1 января 1970 года.

Пример: pe.timestamp > time.now()

1.4 Написание собственных модулей

Начиная с YARA версии 3.0 вы можете расширить ее возможности. Это возможно с помощью модулей, которые вы можете использовать для определения структур данных и функций, которые впоследствии можно будет применить в ваших правилах. Вы можете увидеть некоторые примеры того, что может делать модуль, в Главе 3 Модули.

Цель данной главы состоит в том, чтобы научить вас создавать собственные модули для предоставления YARA дополнительных функций.

1.4.1 Модуль “Hello World!”

Модули пишутся на языке программирования C и встраиваются в YARA в процессе компиляции. Чтобы создавать свои собственные модули, вы должны быть знакомы с языком программирования C, а также с тем, как конфигурировать и собирать YARA из исходного кода. Вам не нужно понимать, как YARA реализует свои функции; YARA предоставляет простой API для модулей, и это все, что вам нужно знать.

Исходный код модуля должен находиться в каталоге libyara/modules с исходными кодами. Рекомендуется использовать имя модуля в качестве имени файла для исходного файла, если имя вашего модуля foo, его исходный файл должен быть foo.c.

В каталоге libyara/modules вы найдете файл demo.c, который мы будем использовать в качестве отправной точки. Файл выглядит следующим образом:

```
#include <yara/modules.h>
#define MODULE_NAME demo
begin_declarations;

    declare_string("greeting");

end_declarations;

int module_initialize(YR_MODULE* module)
{
    return ERROR_SUCCESS;
}

int module_finalize(YR_MODULE* module)
{
    return ERROR_SUCCESS;

int module_load(YR_SCAN_CONTEXT* context, YR_OBJECT* module_object, void* module_data, size_t␣
↪module_data_size)
{
    set_string("Hello World!", module_object, "greeting");
    return ERROR_SUCCESS;
}
int module_unload(YR_OBJECT* module_object)
{
    return ERROR_SUCCESS;
}
#undef MODULE_NAME
```

Начнем разбирать исходный код, чтобы вы могли понять каждую деталь. Первая строка в коде:

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

В заголовочном файле modules.h находятся определения API для модулей YARA, поэтому эта директива include необходима во всех ваших модулях. Вторая строка:

```
#define MODULE_NAME demo
```

Это, определение имени вашего модуля. Для каждого модуля необходимо определить свое имя в начале исходного кода. Имена модулей должны быть уникальными среди модулей, встроенных в YARA.

Затем следует раздел объявлений функций и данных:

```
begin_declarations;

    declare_string("greeting");

end_declarations;
```

Здесь модуль объявляет функции и структуры данных, которые будут доступны для ваших правил YARA. В этом случае мы объявляем только строковую переменную с именем greeting. Более подробно мы обсудим эти вопросы в разделе 4.2.

После раздела объявлений, показанного выше, идет пара функций:

```
int module_initialize(YR_MODULE* module)
{
    return ERROR_SUCCESS;
}

int module_finalize(YR_MODULE* module)
{
    return ERROR_SUCCESS;
}
```

Функция module_initialize вызывается во время инициализации YARA, в то время как функция module_finalize вызывается при завершении YARA. Эти функции позволяют инициализировать и завершить любую глобальную структуру данных, использование которой требуется для работы модуля.

Затем идет функция module_load:

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

Эта функция вызывается один раз для каждого сканируемого файла, но только если модуль импортируется в какое-либо правило с помощью директивы import. Функция module_load позволяет модулю проверять сканируемый файл, разбирать и анализировать его, а затем заполнять структуры данных, определенные в разделе объявлений.

В этом примере функция module_load вообще не проверяет содержимое файла, она просто присваивает строку “Hello World!”к переменной greeting, объявленной ранее.

И, наконец, у нас есть функция module_unload:

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

Для каждого вызова module_load существует соответствующий вызов module_unload. Эта функция позволяет модулю освободить любой ресурс, выделенный во время module_load. В нашем случае ничего не нужно освобождать, поэтому функция просто возвращает ERROR_SUCCESS. И module_load и module_unload должны возвращать ERROR_SUCCESS, чтобы указать, что все прошло нормально. Если возвращается другое значение, сканирование будет прервано и пользователю будет сообщено об ошибке.

Сборка нашего “Hello World!”

Для того, чтобы встроить модули в YARA, необходимо поместить их исходный код в каталог libyara/modules, и выполнить два дальнейших шага, чтобы заставить их работать. Первым шагом является добавление модуля в файл module_list, который также находится в каталоге libyara/modules.

Файл module_list выглядит следующим образом:

```
MODULE(tests)
MODULE(pe)

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

Второй шаг-изменение файла Makefile.am чтобы сообщить программе make, что исходный код вашего модуля должен быть скомпилирован и связан с YARA. В самом начале файла libyara/Makefile.am вы найдете следующее:

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

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

Просто добавьте новую строку для вашего модуля:

И это все! Теперь вы готовы построить YARA с вашим новым модулем. Просто перейдите в корневой каталог с исходниками и введите:

```
make
sudo make install
```

Теперь вы можете создать такое правило:

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

Любой файл, отсканированный с помощью этого правила, будет ему соответствовать, поскольку условие demo.greeting == "Hello World!" всегда true.

1.4.2 Раздел объявлений

В разделе объявлений объявляются переменные, структуры и функции, которые будут доступны для правил YARA. Каждый модуль должен содержать это раздел, который выглядит следующим образом:

```
begin_declarations;

        <your declarations here>

end_declarations;
```

Основные типы

В разделе объявлений можно использовать declare_string(<имя переменной>), declare_integer(<имя переменной>) и declare_float(<имя переменной>) для объявления строковых, целочисленных переменных или переменных с плавающей запятой соответственно. Например:

```
begin_declarations;

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

end_declarations;
```

Note: Переменные с плавающей запятой требуют YARA версии 3.3.0 или более поздней.

Имена переменных могут содержать такие символы как; буквы, цифры и символы подчеркивания. Эти переменные могут быть использованы позже в ваших правилах в любом месте, где ожидается число или строка. Предположим, что имя вашего модуля mymodule, тогда переменные могут быть использованы следующим образом:

```
mymodule.foo > 5

mymodule.bar matches /someregexp/
```

Структуры

Ваши объявления могут быть организованы более структурированным образом:

```
begin_declarations;

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

    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");
        declare_float("tux");

    end_struct("another_structure");

end_declarations;
```

В этом примере мы используем begin_struct(<имя структуры>) и end_struct (<имя структуры>) для разграничения двух структур some_structure и another_structure. В разделители структуры можно по-

местить любые другие объявления, включая другое объявление структуры. Также обратите внимание, что члены разных структур могут иметь одно и то же имя, но члены одной структуры должны иметь уникальные имена.

Обращение к этим переменным из ваших правил будет выглядеть следующим образом:

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

Массивы

Точно так же, как вы объявляете отдельные строки, целые числа, числа с плавающей запятой или структуры, вы можете объявлять массивы из них:

```
begin_declarations;

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

    begin_struct_array("struct_array");

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

    end_struct_array("struct_array");

end_declarations;
```

К отдельным значениям в массиве обращаются, как и в большинстве языков программирования:

```
foo[0]
bar[1]
baz[3]
struct_array[4].foo
struct_array[1].bar
```

Массивы начинаются с нуля и не имеют фиксированного размера, они будут увеличиваться по мере необходимости, когда вы начнете инициализировать его значения.

Словари

Добавлено в версии 3.2.0.

Вы также можете объявить словари целых чисел, чисел с плавающей запятой, строк или структур:

```
begin_declarations;

    declare_integer_dictionary("foo");
    declare_string_dictionary("bar");
    declare_float_dictionary("baz")

    begin_struct_dictionary("struct_dict");
```

(continues on next page)

(continued from previous page)

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

        end_struct_dictionary("struct_dict");

end_declarations;
```

Отдельные значения в словаре доступны с помощью строкового ключа:

```
foo["somekey"]
bar["anotherkey"]
baz["yetanotherkey"]
struct_dict["k1"].foo
struct_dict["k1"].bar
```

Функции

Одной из наиболее мощных возможностей модулей YARA является возможность объявления функций, которые впоследствии могут быть вызваны из ваших правил. Функции должны появляться в разделе объявлений следующим образом:

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

<function name> - это имя, которое будет использоваться в ваших правилах YARA для вызова функции.

<argument types> - это строка, содержащая один символ на аргумент функции, где символ указывает тип аргумента. Функции могут принимать четыре различных типа аргументов: строка, целое число, число с плавающей точкой и регулярное выражение, обозначаемые символами: s, i, f и r соответственно. Если ваша функция в качестве аргумента получает два целых числа, <argument types> должен быть "ii", если она получает целое число в качестве первого аргумента и строку в качестве второго, то <argument types> должен быть "is", если она получает три строки и число с плавающей запятой <argument types> должен быть "sssf".

<return tuype> - это строка с одним символом, обозначающим тип возвращаемого значения. Возможные типы возвращаемых значений: строка "s", целое число "i" и число с плавающей запятой "f".

<C function> - идентификатор для фактической реализации вашей функции.

Ниже приведен полный пример:

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

        return_integer(a + b);
}

define_function(fsum)
{
        double a = float_argument(1);
        double b = float_argument(2);

        return_integer(a + b);
```

(continues on next page)

(continued from previous page)

```
    }

    begin_declarations;

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

end_declarations;
```

Как вы можете видеть в приведенном выше примере, ваш код функции должен быть определен перед разделом объявлений, например:

```
define_function(<function identifier>)
{
    //..ваш код
}
```

Функции могут быть перегружены, как в C++ и других языках программирования. Вы можете объявить две функции с одинаковыми именами, если они различаются по типу или количеству аргументов. Один пример перегруженных функций можно найти в модуле Hash, он имеет две функции для вычисления MD5-хэшей, одна получает в качестве аргументов смещение и длину в файле, а другая получает строку:

```
begin_declarations;

        declare_function("md5", "ii", "s", data_md5);
        declare_function("md5", "s", "s", string_md5);

end_declarations;
```

Подробнее обсудим реализацию функций в разделе 4.5 Подробнее о функциях.

1.4.3 Инициализация и завершение

Каждый модуль должен реализовать две функции для инициализации и завершения: module_initialize и module_finalize. Первый вызывается во время инициализации YARA через функцию yr_initialize () (см. п. 7.6.2), а второй-во время завершения через функцию yr_finalize() (см. п. 7.6.2). Обе функции вызываются независимо от того, импортируется ли модуль каким-либо правилом.

Эти функции дают модулю возможность инициализировать любую глобальную структуру данных, которая ему может понадобиться, но в большинстве случаев это просто пустые функции:

```
int module_initialize(YR_MODULE* module)
{
        return ERROR_SUCCESS;
}

int module_finalize(YR_MODULE* module)
{
        return ERROR_SUCCESS;
}
```

Любое возвращаемое значение, отличное от ERROR_SUCCESS, прервет выполнение YARA.

1.4.4 Реализации логики работы модуля

Кроме module_initialize и module_finalize каждый модуль должен реализовывать еще две функции, которые вызываются YARA при сканировании файла или пространства памяти процесса: module_load и module_unload. Обе функции вызываются один раз для каждого сканируемого файла или процесса, но только если модуль был импортирован с помощью директивы import. Если модуль не импортируется в какое-либо правило, то module_load или module_unload вызываться не будут.

Функция module_load имеет следующий прототип:

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

Аргумент context содержит информацию относительно текущего сканирования, включая сканируемые данные. Аргумент module_object является указателем на структуру YR_OBJECT, связанную с модулем. Каждая структура, переменная или функция, объявленная в модуле YARA, представлена структурой YR_OBJECT. Эти структуры образуют дерево, корнем которого является структура модуля YR_OBJECT. Например, если у вас есть следующие объявления в модуле с именем mymodule:

```
begin_declarations;

      declare_integer("foo");

            begin_struct("bar");

                  declare_string("baz");

            end_struct("bar");

end_declarations;
```

Тогда дерево будет выглядеть так:

```
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")
```

Обратите внимание, что и bar, и mymodule имеют одинаковый тип OBJECT_TYPE_STRUCT, что означает, что YR_OBJECT, связанный с модулем, является просто еще одной структурой, подобной bar. Фактически, когда вы пишете в своих правилах что-то вроде mymodule.foo, вы выполняете поиск полей в структуре так же, как это делает bar.baz.

Таким образом, аргумент module_object позволяет вам получить доступ к каждой переменной, структуре или функции, объявленной модулем, предоставив указатель на корень дерева объектов.

Аргумент module_data - это указатель на любые дополнительные данные, передаваемые модулю, а module_data_size - это размер этих данных. Не все модули требуют дополнительных данных, большинство из них полагаются только на данные, которые сканируются, но некоторые из них требуют дополнительной информации в качестве входных данных. Модуль Cuckoo является хорошим примером этого, он получает отчет о поведении, связанный с проверяемыми PE-файлами, который передается в аргументах module_data и module_data_size.

Для получения дополнительной информации о том, как передать дополнительные данные в ваш модуль, посмотрите на применение опции -x в Главе 5.

Доступ к сканируемым данным

Большинству модулей YARA необходим доступ к сканируемому файлу или памяти процесса, чтобы извлечь из него информацию. Сканируемые данные отправляются в модуль в структуре YR_SCAN_CONTEXT, передаваемой в функцию mdule_load. Данные иногда разбиваются на блоки, поэтому вашему модулю необходимо выполнять итерации по блокам с помощью макроса foreach_memory_block:

```
int module_load(
        R_SCAN_CONTEXT* context,
        YR_OBJECT* module_object,
        void* module_data,
        size_t module_data_size)
{
        YR_MEMORY_BLOCK* block;
        foreach_memory_block(context, block)
        {
                //..делаем какие-либо операции с текущим блоком памяти
        }
}
```

Каждый блок памяти представлен структурой YR_MEMORY_BLOCK со следующими атрибутами:

- YR_MEMORY_BLOCK_FETCH_DATA_FUNC fetch_data

Указатель на функцию, возвращающую указатель на блок данных.

- size_t size

Размер блока данных.

- size_t base

Базовое смещение / адрес для этого блока. Если файл сканируется, это поле содержит смещение в файле, с которого начинается блок, если сканируется область памяти процесса, он содержит виртуальный адрес, с которого начинается блок.

Блоки всегда повторяются в том же порядке, в котором они появляются в файле или в памяти процесса. В случае файлов первый блок будет содержать начало файла. Фактически, в большинстве случаев один блок будет содержать содержимое всего файла, но вы не можете полагаться на это при написании кода. Для очень больших файлов YARA может в конечном итоге разбить файл на два или более блоков, и ваш модуль должен быть готов к этому.

При сканировании пространства памяти процесса ваш модуль определенно получит большое количество блоков, по одному для каждой выделенной области памяти в адресном пространстве процесса.

Однако в некоторых случаях перебирать блоки не требуется. Если ваш модуль просто анализирует заголовок какого-либо формата файла, вы можете смело предполагать, что весь заголовок содержится в первом блоке (тем не менее, добавьте некоторые проверки в ваш код). В этих случаях вы можете использовать макрос first_memory_block:

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

(continues on next page)

(continued from previous page)

```
{
        YR_MEMORY_BLOCK* block;
        const uint8_t* block_data;

        block = first_memory_block(context);
        block_data = block->fetch_data(block)

        if (block_data != NULL)
        {
                //..делаем какие-либо операции с текущим блоком памяти
        }
}
```

В предыдущем примере вы также можете увидеть, как использовать функцию fetch_data. Эта функция, которая является членом структуры YR_MEMORY_BLOCK, получает указатель на тот же блок и возвращает указатель на данные блока. Вашему модулю не принадлежит память, на которую указывает этот указатель, освобождение этой памяти не является вашей ответственностью. Однако имейте в виду, что указатель действителен только до тех пор, пока вы не запросите следующий блок памяти. Пока вы используете указатель в пределах foreach_memory_block, вы в безопасности. Также учтите, что fetch_data может возвращать указатель NULL, ваш код должен быть подготовлен для этого случая.

```
const uint8_t* block_data;

foreach_memory_block(context, block)
{
        block_data = block->fetch_data(block);

        if (block_data != NULL)
        {
                // использование block_data здесь безопасно.
        }
}
// память, на которую указывает block_data, здесь уже может быть освобождена.
```

Присваивание значений переменным

Функция module_load позволяет назначать значения переменным, объявленным в разделе объявлений, после того, как вы пропарсили или проанализировали сканируемые данные и/или данные любого дополнительного модуля. Это делается с помощью функций set_integer и set_string:

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

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

Обе функции получают значение, которое должно быть присвоено переменной, указатель на YR_OBJECT, представляющий саму переменную или некоторого предка этой переменной, дескриптор поля и дополнительные аргументы, как определено дескриптором поля.

Если мы присваиваем значение переменной, представленной самим объектом, то дескриптор поля должен быть NULL.

Например, предполагая, что объект указывает на структуру YR_OBJECT, соответствующую некоторой целочисленной переменной, мы можем установить значение для этой целочисленной переменной с помощью:

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

Дескриптор поля используется, когда вы хотите присвоить значение некоторому потомку объекта. Например, рассмотрим следующие объявления:

```
begin_declarations;

    begin_struct("foo");

        declare_string("bar");

        begin_struct("baz");

            declare_integer("qux");

        end_struct("baz");

    end_struct("foo");

end_declarations;
```

Если объект указывает на YR_OBJECT, связанный со структурой foo, вы можете установить значение для строки bar следующим образом:

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

И значение для qux таким образом:

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

Вы помните, что аргумент module_object для module_load был указателем на YR_OBJECT? Вы помните, что этот YR_OBJECT является структурой, как и bar? Исходя из этого, вы также можете установить значения для bar и qux следующим образом:

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

Но что происходит с массивами? Каким образом можно установить значения для элементов массива? Если у вас есть следующее объявление:

```
begin_declarations;

    declare_integer_array("foo");

    begin_struct_array("bar")

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

    end_struct_array("bar");

end_declarations;
```

Тогда следующие представления операторов set_integer и set_struing являются валидными:

```
set_integer(<value>, module, "foo[0]");
set_integer(<value>, module, "foo[%i]", 2);
```

(continues on next page)

(continued from previous page)

```
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);
```

Спецификатор формата %i в дескрипторе поля заменяются дополнительными целочисленными аргументами, передаваемыми функции. Это работает так же, как printf в программах на C, но единственными допустимыми спецификаторами формата являются %i и %s для целочисленных и строковых аргументов соответственно.

Спецификатор формата %s используется для назначения значений определенному ключу в словаре:

```
set_integer(<value>, module, "foo[\"key\"]");
set_integer(<value>, module, "foo[%s]", "key");
set_string(<value>, module, "bar[%s].baz", "another_key");
```

Если явно не присвоить значение объявленной переменной, массиву или элементу справочника, то они останутся в неопределенном состоянии. Это не проблема, и даже полезно во многих случаях. Например, если модуль предназначен для анализа файлов определенного формата, а получает для анализа файлы другого формата, можно оставить все переменные неопределенными, а не присваивать им фиктивные значения, которые не имеют смысла. YARA будет обрабатывать неопределенные значения в условиях правила, как описано в Главе 3 Модули.

В дополнение к функциям set_integer и set_string у вас есть их аналоги get_integer и get_string. Как следует из их имен, они используются для получения значения переменной, что может быть полезно при реализации ваших функций для получения значений, ранее сохраненных в module_load.

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

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

Также есть функция для получения любого YR_OBJECT в дереве объектов:

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

Теперь небольшой экзамен...

Эквивалентны ли следующие две строки? Почему?

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

Сохранение данных для дальнейшего использования

Иногда информации, хранящейся непосредственно в ваших переменных, записанных с помощью set_integer и set_string, недостаточно. Возможно, вам потребуется хранить более сложные структуры данных или информацию, которую не нужно предоставлять правилам YARA.

Хранение информации важно, когда ваш модуль экспортирует функции для использования в правилах YARA. Реализация этих функций обычно требует доступа к информации, генерируемой module_load, которая должна где-то храниться. У вас может возникнуть желание определить глобальные переменные для хранения необходимой информации, но это сделает ваш код не поточно-ориентированным. Правильный подход заключается в использовании поля данных структур YR_OBJECT.

Каждый YR_OBJECT имеет поле void* data, которое может быть безопасно использовано вашим кодом для хранения указателя на любые данные, которые вам могут понадобиться. Типичный шаблон использует поле data YR_OBJECT модуля, как в следующем примере:

```
typedef struct _MY_DATA
{
    int some_integer;

} MY_DATA;

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

    return ERROR_SUCCESS;
}
```

Не забудьте освободить выделенную память в функции module_unload:

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

    eturn ERROR_SUCCESS;
}
```

Note: Не используйте глобальные переменные для хранения данных. Функции в модуле могут быть вызваны из разных потоков одновременно, и может произойти повреждение данных или неправильное поведение.

1.4.5 Подробнее о функциях

Мы уже показали, как объявить функцию в разделе объявлений (см. п. 4.2.5). Здесь мы собираемся показать, как обеспечить их реализацию.

Аргументы функций

В коде функции вы получаете ее аргументы с помощью integer_argument(n), float_argument(n), regexp_argument(n), string_argument(n) или sized_string_argument(n) в зависимости от типа аргумента, где n - номер аргумента начиная с 1.

string_argument(n) может использоваться, когда ваша функция ожидает получить C-строку с нулевым завершением, если ваша функция может получать произвольные двоичные данные, возможно содержащие нулевые байты, вы должны использовать sized_string_argument(n).

Вот несколько примеров:

```
int64_t arg_1 = integer_argument(1);
RE* arg_2 = regexp_argument(2);
char* arg_3 = string_argument(3);
SIZED_STRING* arg_4 = sized_string_argument(4);
double arg_5 = float_argument(1);
```

Тип C для целочисленных аргументов - int64_t, для аргументов с плавающей запятой - double, для регулярных выражений - RE*, для NULL-завершенных строк - char*, а для строк, возможно содержащих NULL-символы, - SIZED_STRING*. Структуры SIZED_STRING имеют следующие атрибуты:

SIZED_STRING

- length - Длина строки.
- c_string - char* указатель на содержимое строки.

Возвращаемые значения

Функции могут возвращать три типа значений: строки, целые числа и числа с плавающей точкой. Вместо использования оператора возврата, используемого в языке программирования C вы должны использовать return_string (x), return_integer (x) или return_float (x) для возврата из функции, в зависимости от типа возвращаемого значения функции. Во всех случаях x является константой, переменной или выражением, оцениваемым как char*, int64_t или double соответственно.

Вы можете использовать return_string (UNDEFINED), return_float (UNDEFINED) и return_integer (UNDEFINED) для возврата неопределенных значений из функции. Это полезно во многих ситуациях, например, если аргументы, переданные функциям, не имеют смысла, или если ваш модуль ожидает определенный формат файла, а сканируемый файл - другого формата, или в любом другом случае, когда ваша функция не может возвратить верное значение.

Note: Не используйте оператор возврата C для возврата из функции. Возвращаемое значение будет интерпретировано как код ошибки.

Доступ к объектам

При написании функции нам иногда требуется доступ к значениям, ранее назначенным переменным модуля, или дополнительным данным, хранящимся в поле data структур YR_OBJECT, как обсуждалось ранее в п. 4.4.3, для последующего использования. Но для этого нам нужен способ, позволяющий получить доступ к соответствующей структуре YR_OBJECT. Для этого есть две функции: module () и parent (). Функция module () возвращает указатель на YR_OBJECT верхнего уровня, соответствующий модулю, который передается в функцию module_load. Функция parent () возвращает указатель на YR_OBJECT, соответствующий структуре, в которой содержится функция. Например, рассмотрим следующий фрагмент кода:

```
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;
}
```

(continues on next page)

(continued from previous page)

```
begin_declarations;

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

    begin_struct("foo");

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

    end_struct("foo");

end_declarations;
```

В функции f1 переменная module указывает на верхний уровень YR_OBJECT, а также на переменную parent, потому что родителем для f1 является сам модуль. Однако в функции f2 переменная parent указывает на YR_OBJECT, соответствующий структуре foo, а module указывает на верхний уровень YR_OBJECT, как и в первом случае.

Контекст сканирования

Из функции вы также можете получить доступ к структуре YR_SCAN_CONTEXT, обсуждавшейся ранее в п. 4.4.1. Это полезно для функций, которые должны проверять сканируемый файл или память процесса. Вот как вы получаете указатель на структуру YR_SCAN_CONTEXT:

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

1.5 Запуск YARA из командной строки

Чтобы вызвать YARA, вам понадобятся две вещи: файл с правилами (RULES_FILE), которые вы хотите использовать (либо в исходном коде, либо в скомпилированной форме), и цель для сканирования (TARGET). Целью может быть файл, папка или процесс.

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

RULES_FILE может быть передан непосредственно в форме исходного кода или может быть предварительно скомпилирован с помощью инструмента yarac. Вы можете предпочесть использовать свои правила в скомпилированной форме, если вы собираетесь вызывать YARA несколько раз с одними и теми же правилами. Таким образом, вы сэкономите время, потому что для YARA быстрее загружать скомпилированные правила, заново чем компилировать одни и те же правила.

Вы также можете передать несколько исходных файлов в yara, как в следующем примере:

```
yara [OPTIONS] RULES_FILE_1 RULES_FILE_2 RULES_FILE_3 TARGET
```

Однако обратите внимание, что это работает только для правил в исходной форме. При вызове YARA с скомпилированными правилами принимается только один файл с правилами.

В приведенном выше примере все правила имеют одно и то же пространство имен по умолчанию, что означает, что идентификаторы правил должны быть уникальными среди всех файлов. Однако можно указать пространство имен для отдельных файлов. Например:

```
yara [OPTIONS] namespace1:RULES_FILE_1 RULES_FILE_2 RULES_FILE_3 TARGET
```

В этом случае RULE_FILE_1 использует пространство имен namespace1, а rules_file_2 и RULES_FILE_3 используют пространство имен по умолчанию.

Во всех случаях правила будут применяться к цели, указанной в качестве последнего аргумента для YARA, если это путь к каталогу, то все файлы, содержащиеся в нем, будут проверены. По умолчанию YARA не сканирует каталоги рекурсивно, для этого можно использовать опцию -r.

Доступные опции:

-t <tag> –tag=<tag>

Вывод правила с тегом <tag> и игнорирование остальных правил.

-i <identifier> –identifier=<identifier>

Вывод правила с именем <identifier> и игнорирование остальных правил.

-c –count

Вывод только нужного количества совпадений.

-n

Печатать только невыполненных правил (отрицание).

-D –print-module-data

Вывод данных модуля.

-g –print-tags

Вывод тегов.

-m –print-meta

Вывод метаданных.

-s –print-strings

Вывод совпадающих строк.

-L –print-string-length

Вывод длины совпадающих строк.

-e –print-namespace

Вывод пространства имен правил.

-p <number> –threads=<number>

Использование указанного числа потоков <number> для сканирования каталога.

-l <number> –max-rules=<number>

Прервать сканирование после совпадения нескольких <number> правил.

-a <seconds> –timeout=<seconds>

Прервать сканирование по истечении нескольких <seconds> секунд.

-k <slots> –stack-size=<slots>

Выделение стека нужного размер для необходимого количества слотов <slots>. По умолчанию: 16384. Это позволит использовать более объемные правила, хотя и с большим объемом памяти.

Добавлено в версии 3.5.0.

–max-strings-per-rule=<number>

Установка максимального числа строк в правиле (по умолчанию=10000). Если в правиле больше указанного числа строк, возникает ошибка.

Добавлено в версии 3.7.0.

-d <identifier>=<value>

Определить внешнюю переменную.

-x <module>=<file>

Передать содержимое файла <file> в качестве дополнительных данных в модуль <module>.

-r –recursive

Рекурсивное сканирование каталога.

-f –fast-scan

Режим быстрого соответствия.

-w –no-warnings

Отключить предупреждения

–fail-on-warnings

Обрабатывать предупреждения как ошибки. Не имеет эффекта, если используется с --no-warnings.

-v –version

Показать информацию о версии.

-h –help

Показать справку.

Вот несколько примеров:

Применить правило в /foo/bar/rules ко всем файлам в текущем каталоге. Подкаталоги не сканируются:

```
yara /foo/bar/rules
```

Применить правила в /foo/bar/rules к bazfile. Только отчеты о правилах, помеченных как Packer или Compiler:

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

Сканирование всех файлов в каталоге /foo и его подкаталогах:

```
yara -r /foo
```

Определение трех внешних переменных mybool, myint и mystring:

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

Применить правила в /foo/bar/rules для bazfile при передаче содержимого cuckoo_json_report к модулю cuckoo:

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

1.6 Использование YARA из Python

Функции YARA могут быть использованы в Python-скриптах с помощью библиотеки yara-python. Как только библиотека будет построена и установлена, как описано в п. 1.1, вы получите доступ к полному потенциалу YARA из ваших скриптов Python. Первый шаг-импорт библиотеки YARA:

```
import yara
```

Затем вам нужно будет скомпилировать ваши правила YARA, прежде чем применять их к вашим данным:

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

Аргумент по умолчанию - filepath, поэтому вам не нужно явно указывать его имя:

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

Вы также можете скомпилировать свои правила из файлового объекта:

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

Или вы можете скомпилировать их непосредственно из строки Python:

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

Если вы хотите скомпилировать группу файлов или строк одновременно, вы можете сделать это, используя filepaths или sources в наименовании аргумента:

```
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 }'
})
```

Обратите внимание, что и filepaths, и sources должны быть словарями с ключами строкового типа. Ключи словаря используются в качестве идентификатора пространства имен, позволяющего различать правила с одинаковыми именами в разных источниках, как это происходит во втором примере с именем dummy.

Метод compile также имеет необязательный логический параметр с именем include, который позволяет вам контролировать, следует ли принимать директиву include в исходных файлах, например:

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

Если исходный файл содержит директивы include, показанный пример вызовет исключение.

Если используется include, можно установить обратный вызов Python, чтобы определить собственный источник для импортируемых файлов (по умолчанию они читаются с диска). Эта функция обратного вызова устанавливается с помощью необязательного параметра include_callback. Он получает следующие параметры:

- requested_filename: файл, запрошенный с помощью include
- filename: файл, содержащий директиву include, если применимо, иначе None
- namespace: пространство имен

И возвращает запрошенные источники правил в виде одной строки.

Если вы используете внешние переменные в своих правилах, вы должны определить эти внешние переменные либо при компиляции правил, либо при применении правил к некоторому файлу. Чтобы определить ваши переменные в момент компиляции, вы должны передать параметр externals методу compile. Например:

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

Параметр externals должен представлять собой словарь с именами переменных в качестве ключей и связанным значением типа строка, целое число или логическая переменная.

Метод compile также принимает необязательный логический аргумент error_on_warning. Этот аргумент указывает YARA давать исключение при выдаче предупреждения во время компиляции. Такие

предупреждения обычно выдаются, когда ваши правила содержат некоторую конструкцию, которая может замедлять сканирование. Значение по умолчанию для аргумента error_on_warning - false.

Во всех случаях compile возвращает экземпляр класса yara.Rules Rules. Этот класс имеет метод save, который можно использовать для сохранения скомпилированных правил в файл:

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

Скомпилированные правила могут быть загружены позже с помощью метода load:

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

Начиная с YARA 3.4, save и load принимают файловые объекты. Например, вы можете сохранить ваши правила в буфере памяти с помощью этого кода:

```
import StringIO

buff = StringIO.StringIO()
rules.save(file=buff)
```

Сохраненные правила могут быть загружены из буфера памяти:

```
buff.seek(0)
rule = yara.load(file=buff)
```

Результатом загрузки также является экземпляр класса yara.Rules.

Экземпляры Rules также имеют метод match, который позволяет применять правила к файлу:

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

Но вы также можете применить правила к строке Python:

```
with open('/foo/bar/my_file', 'rb') as f:
    matches = rules.match(data=f.read())
```

Или к запущенному процессу:

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

Как и в случае с compile, метод match может получать определения для внешних переменных в аргументе externals.

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

Внешние переменные, определенные во время компиляции, не требуют повторного определения при последующих вызовах метода match. Однако вы можете переопределить любую переменную по мере необходимости или сделать дополнительные определения, которые не были сделаны во время компиляции.

В некоторых ситуациях, связанных с очень большим набором правил или большими файлами, метод match может занять слишком много времени для запуска. В этих ситуациях вам может пригодиться аргумент timeout:

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

Если match не завершается до истечения указанного количества секунд, возникает исключение TimeoutError.

Вы также можете указать функцию обратного вызова при вызове метода match. По умолчанию предоставленная функция будет вызываться для каждого правила, независимо от того, соответствует оно или нет. Вы можете выбрать, когда вызывается ваша функция обратного вызова, установив для параметра which_callbacks одно из значений: yara.CALLBACK_MATCHES, yara.CALLBACK_NON_MATCHES или yara.CALLBACK_ALL. По умолчанию используется yara.CALLBACK_ALL. Ваша функция обратного вызова должна ожидать один параметр типа dictionary и должна возвращать CALLBACK_CONTINUE для перехода к следующему правилу или CALLBACK_ABORT, чтобы прекратить применять правила к вашим данным.

Например:

```
import yara

def mycallback(data):
        print data
        return yara.CALLBACK_CONTINUE

matches = rules.match('/foo/bar/my_file', callback=mycallback, which_callbacks=yara.CALLBACK_
↪MATCHES)
```

Переданный словарь будет примерно таким:

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

Поле matches указывает, соответствует ли правило данным или нет. Поля strings - это список совпадающих строк с векторами вида:

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

Метод match возвращает список экземпляров класса yara.Match. Экземпляры этого класса имеют те же атрибуты, что и словарь, передаваемый в функцию обратного вызова.

Вы также можете указать функцию обратного вызова модуля при вызове метода match. Предоставленная функция будет вызываться для каждого импортированного модуля, который сканировал файл. Ваша функция обратного вызова должна ожидать один параметр типа словаря и должна возвращать CALLBACK_CONTINUE для перехода к следующему правилу или CALLBACK_ABORT, чтобы прекратить применять правила к вашим данным.

Например:

```
import yara

def modules_callback(data):
        print data
        return yara.CALLBACK_CONTINUE

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

Переданный словарь будет содержать информацию из модуля.

Вы также можете обнаружить, что размеры по умолчанию для стека для соответствующего механизма в YARA или размер по умолчанию для максимального количества строк в правиле слишком

малы. В libyara API вы можете изменить их, используя переменные YR_CONFIG_STACK_SIZE и YR_CONFIG_MAX_STRINGS_PER_RULE через функцию yr_set_configuration в libyara. Инструмент командной строки предоставляет их в качестве аргументов командной строки --stack-size (-k) и --max-strings-per-rule. Чтобы установить эти значения через API-интерфейс Python, вы можете использовать yara.set_config с любым из двух stack_size и max_strings_per_rule или обоими, указанными как kwargs. На момент написания этой статьи размер стека по умолчанию составлял 16384, а максимальное число строк по умолчанию для правила составляло 10000.

Например:

```
yara.set_config(stack_size=65536)
yara.set_config(max_strings_per_rule=50000, stack_size=65536)
yara.set_config(max_strings_per_rule=20000)
```

1.6.1 Описание

yara. compile(. . .)

Компиляция правил YARA из исходников.

Должны быть указаны filepath, source, file, filepaths или sources. Остальные аргументы являются необязательными.

Параметры:

- filepath (str) – Путь к исходному файлу.
- source (str) – Строка, содержащая код правил.
- file (file-object) – Исходный файл в виде файлового объекта.
- filepaths (dict) – Словарь, где ключи - это пространства имен, а значения - пути к исходным файлам.
- sources (dict) – Dictionary where keys are namespaces and values are strings containing rules code.
- externals (dict) – Словарь с внешними переменными. Ключи - это имена переменных, а значения - значения переменных.
- includes (boolean) – true, если директивы include разрешены, или false в противном случае. Значение по умолчанию: true.
- error_on_warning (boolean) – Если предупреждения рассматриваются как ошибки, возникает исключение.

Возвращает: Скомпилированный объект правил.

Возвращаемый тип: yara.Rules

Исключения

- YaraSyntaxError – Если была обнаружена синтаксическая ошибка.
- YaraError – Если произошла ошибка.

---

yara. load(. . .)

Изменено в версии 3.4.0.

Загрузка скомпилированных правил из пути или файлового объекта. Необходимо указать либо filepath, либо file.

Параметры:

- filepath (str) – Путь к файлу с скомпилированными правилами.
- file (file-object) – Файловый объект, поддерживающий метод read.

Возвращает: Скомпилированный объект правил.

Возвращаемый тип: yara.Rules.

Исключения: YaraError - Если при загрузке файла произошла ошибка.

---

yara. set_config(...)

Установка переменных конфигурации, доступных через API yr_set_configuration.

Укажите либо stack_size, либо max_strings_per_rule. Эти kwargs принимают целочисленные значения без знака в качестве входных данных и присваивают предоставленное значение переменным в yr_set_configuration (...) - YR_CONFIG_STACK_SIZE и YR_CONFIG_MAX_STRINGS_PER_RULE соответственно.

Праметры:

- stack_size (int) – Размер стека, используемый для YR_CONFIG_STACK_SIZE.
- max_strings_per_rule (int) – Максимальное количество строк, разрешенное для правила YARA. Будет сопоставлен с YR_CONFIG_MAX_STRINGS_PER_RULE.

Возвращает: - .

Возвращаемый тип: - .

Исключения: YaraError - Если произошла ошибка.

---

class yara. Rules

Экземпляры этого класса возвращаются функцией yara.compile () и представляют собой набор скомпилированных правил.

match (filepath, pid, data, externals=None, callback=None, fast=False, timeout=None, modules_data=None, modules_callback=None, which_callbacks=CALLBACK_ALL)

Сканирование файла, памяти процесса или строки данных.

Должны быть указаны filepath, pid или data. Остальные аргументы являются необязательными.

Параметры:

- filepath (str) – Путь к сканируемому файлу.
- pid (int) – Идентификатор сканируемого процесса.
- data (str) – Сканируемые данные.
- externals (dict) – Словарь с внешними переменными. Ключи - это имена переменных, а значения - значения переменных.
- callback (function) – Функция обратного вызова, вызываемая для каждого правила.
- fast (bool) – Если true, выполняется сканирование в быстром режиме.
- timeout (int) – Прерывает сканирование, если количество указанных секунд истекло.
- modules_data (dict) – Словарь с дополнительными данными к модулям. Ключи - это имена модулей, а значения - байтовые объекты, содержащие дополнительные данные.

- modules_callback (function) – Функция обратного вызова, вызываемая для каждого модуля.
- which_callbacks (int) – Целое число, которое указывает, в каких случаях должна вызываться функция обратного вызова. Возможные значения: yara.CALLBACK_ALL, yara.CALLBACK_MATCHES и yara.CALLBACK_NON_MATCHES.

Исключения:

- YaraTimeoutError – Если тайм-аут был достигнут.
- YaraError – Если во время сканирования произошла ошибка.

save(. . .)

Изменено в версии 3.4.0.

Сохраняет скомпилированные правила в файл. Необходимо указать либо путь к файлу, либо файл.

Параметры:

- filepath (str) – Путь к файлу.
- file (file-object) – Файловый объект, поддерживающий метод write.

Исключения: YaraError Если при сохранении файла произошла ошибка.

---

class yara. Match

Объекты, возвращаемые yara.match (), представляют совпадения.

rule

Имя совпавшего правила.

namespace

Пространство имен, связанное с совпавшим правилом.

tags

Массив строк, содержащих теги, связанные с совпавшим правилом.

meta

Словарь, содержащий метаданные, связанные с совпавшим правилом.

strings

Список кортежей, содержащих информацию о соответствующих строках. Каждый кортеж имеет форму: (<смещение>, <строковый идентификатор>, <строковые данные>).

1.7 YARA API для C

Вы можете интегрировать YARA в свой C/C++ проект с помощью API, предоставляемого библиотекой libyara. Этот API дает вам доступ к каждой функции YARA, и это тот же API, который используется инструментами командной строки yara и yarac.

1.7.1 Инициализация и завершение libyara

Первое, что ваша программа должна сделать при использовании libyara - это инициализации библиотеки. Это делается путем вызова функции yr_initialize(). Эта функция выделяет все ресурсы, необходимые библиотеке, и инициализирует внутренние структуры данных. В паре с функцией yr_initialize() работает функция yr_finalize(), которая должна быть вызвана, когда вы закончите использовать библиотеку.

В многопоточной программе функции yr_initialize() и yr_finalize() должен вызывать только основной поток. Никаких других действий от других потоков, использующих библиотеку не требуется.

1.7.2 Компиляция правил

Перед использованием ваших правил для сканирования любых данных вам необходимо скомпилировать их в двоичную форму. Для этого вам понадобится компилятор YARA, который можно создать с помощью yr_compiler_create(). После использования компилятор должен быть уничтожен с помощью yr_compiler_destroy().

Вы можете использовать yr_compiler_add_file(), yr_compiler_add_fd() или yr_compiler_add_string(), чтобы добавить один или несколько входных источников для компиляции. Эти функции получают необязательное пространство имен. Правила, добавленные в том же пространстве имен, ведут себя так, как если бы они содержались в одном и том же исходном файле или строке, поэтому идентификаторы правил должны быть уникальными среди всех источников, совместно использующих пространство имен. Если аргумент пространства имен равен NULL, правила помещаются в пространство имен по умолчанию (default).

Функции yr_compiler_add_file(), yr_compiler_add_fd() и yr_compiler_add_string() возвращают количество ошибок, найденных в исходном коде. Если правила верны, они возвращают 0. Если какая-либо из этих функций возвращает ошибку, компилятор больше не может использоваться ни для добавления дополнительных правил, ни для получения скомпилированных правил.

Для получения подробной информации об ошибке вы должны установить функцию обратного вызова, используя yr_compiler_set_callback() перед вызовом любой из функций компиляции. Функция обратного вызова имеет следующий прототип:

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

Изменения в версии 3.3.0.

Возможные значения для error_level: YARA_ERROR_LEVEL_ERROR и YARA_ERROR_LEVEL_WARNING. Аргументы file_name и line_number содержат имя файла и номер строки, где происходит ошибка или предупреждение. file_name - это то, что передается в yr_compiler_add_file() или yr_compiler_add_fd(). Это может быть NULL, если вы передали NULL или если вы используете yr_compiler_add_string(). Указатель user_data - это то же самое, что вы передали в yr_compiler_set_callback().

По умолчанию для правил, содержащих ссылки на другие файлы (включая “filename.yara”), YARA попытается найти эти файлы на диске. Однако, если вы хотите получить импортированные правила из другого источника (например, из базы данных или удаленной службы), функцию обратного вызова можно установить с помощью yr_compiler_set_include_callback().

Обратный вызов получает следующие параметры:

- include_name: имя запрашиваемого файла.
- calling_rule_filename: запрашивающее имя файла (NULL, если не файл).
- calling_rule_namespace: пространство имен (NULL, если не определено).
- user_data: указатель тот же, что был передан в yr_compiler_set_include_callback().

Он должен вернуть содержимое запрошенного файла в виде строки с нулевым символом в конце. Память для этой строки должна быть выделена функцией обратного вызова. Как только можно будет безопасно освободить память, используемую для возврата результата обратного вызова, будет вызвана функция include_free, переданная в yr_compiler_set_include_callback(). Если память не нужно освобождать, вместо нее можно передать NULL. Вы можете полностью отключить поддержку включений, установив функцию обратного вызова в NULL с помощью yr_compiler_set_include_callback().

Функция обратного вызова имеет следующий прототип:

```
const char* include_callback(
        const char* include_name,
        const char* calling_rule_filename,
        const char* calling_rule_namespace,
        void* user_data);
```

Функция освобождения памяти имеет следующий прототип:

```
void include_free(
        const char* callback_result_ptr,
        void* user_data);
```

После успешного добавления некоторых источников можно получить скомпилированные правила с помощью функции yr_compiler_get_rules(). Вы получите указатель на структуру YR_RULES, которая может быть использована для сканирования ваших данных, как описано в разделе сканирование данных. После того, как yr_compiler_get_rules() вызвана, вы не сможете добавить больше источников в компилятор, но вы можете получить несколько экземпляров скомпилированных правил, вызывая yr_compiler_get_rules() несколько раз.

Каждый экземпляр YR_RULES должен быть уничтожен с помощью yr_rules_destroy().

1.7.3 Определение внешних переменных

Если в ваших правилах используются внешние переменные (как в примере ниже), вы должны определить эти переменные, используя любую из функций yr_compiler_define_XXXX_variable. Переменные должны быть определены до того, как правила скомпилированы с помощью функции yr_compiler_add_XXXX, и они должны быть определены с типом, который соответствует контексту, в котором переменная используется в правиле, к примеру, переменная, которая используется как my_var == 5, не может быть определена как строковая переменная.

При определении внешних переменных с помощью yr_compiler_define_XXXX_variable вы должны предоставить значение для каждой переменной. Это значение встраивается в скомпилированные правила и используется всякий раз, когда переменная появляется в правиле. Однако вы можете изменить значение, связанное с внешней переменной, после компиляции правил, используя любую из функций yr_rules_define_XXXX_variable.

1.7.4 Сохранение и извлечение скомпилированных правил

Скомпилированные правила могут быть сохранены в файл и извлечены позже с помощью yr_rules_save() и yr_rules_load(). Правила, скомпилированные и сохраненные на одной машине, мо-

гут быть загружены на другую машину, если они имеют одинаковый порядок байтов, независимо от операционной системы или ее разрядности (32 или 64-разрядная). Однако файлы, сохраненные в более старых версиях YARA, могут не работать с более новыми версиями из-за изменений в макете файла.

Вы также можете сохранять свои правила в общие потоки данных и извлекать их из общих потоков данных с помощью функций yr_rules_save_stream() и yr_rules_load_stream() соответственно. Эти функции получают указатель на структуру YR_STREAM, определенную как:

```
typedef struct _YR_STREAM
{
    void* user_data;

    YR_STREAM_READ_FUNC read;
    YR_STREAM_WRITE_FUNC write;

} YR_STREAM;
```

Вы должны предоставить свои собственные реализации для функций чтения и записи. Функция чтения используется yr_rules_load_stream() для чтения данных из вашего потока, а функция записи используется yr_rules_save_stream() для записи данных в ваш поток.

Ваши реализации функций read и write должны отвечать этим прототипам:

```
size_t read(
    void* ptr,
    size_t size,
    size_t count,
    void* user_data);

size_t write(
    const void* ptr,
    size_t size,
    size_t count,
    void* user_data);
```

Аргумент ptr - это указатель на буфер, куда функция read должна поместить прочитанные данные, или где функция write найдет данные, которые должны быть записаны в поток. В обоих случаях size - это размер каждого читаемого или записываемого элемента и count количества элементов. Общий размер читаемых или записываемых данных равен size * count. Функция read должна возвращать количество прочитанных элементов, функция write должна возвращать общее количество записанных элементов.

Указатель user_data является тем же, который вы указали в структуре YR_STREAM. Вы можете использовать его для передачи произвольных данных в функции чтения и записи.

1.7.5 Сканирование данных

Если у вас есть экземпляр YR_RULES, вы можете использовать его непосредственно с одной из функций yr_rules_scan_XXXX, описанной ниже, или создать сканер с помощью yr_scanner_create (). Давайте начнем с обсуждения первого подхода.

YR_RULES, который вы получили от компилятора, может использоваться с yr_rules_scan_file(), yr_rules_scan_fd() или yr_rules_scan_mem() для сканирования файла, дескриптора файла или буфера в памяти соответственно. Результаты сканирования возвращаются в вашу программу через функцию обратного вызова. Обратный вызов имеет следующий прототип:

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

Возможные значения для message:

```
CALLBACK_MSG_RULE_MATCHING
CALLBACK_MSG_RULE_NOT_MATCHING
CALLBACK_MSG_SCAN_FINISHED
CALLBACK_MSG_IMPORT_MODULE
CALLBACK_MSG_MODULE_IMPORTED
```

Ваша функция обратного вызова будет вызываться один раз для каждого правила с сообщением CALLBACK_MSG_RULE_MATCHING или CALLBACK_MSG_RULE_NOT_MATCHING, в зависимости от того, выполняется правило или нет. В обоих случаях указатель на структуру YR_RULE, связанную с правилом, передается в аргументе message_data. Вам просто нужно выполнить приведение типа из void* в YR_RULE*, чтобы получить доступ к структуре.

Этот обратный вызов также может быть вызван с сообщением CALLBACK_MSG_IMPORT_MODULE. Все модули, на которые ссылается оператор import в правилах, импортируются один раз для каждого сканируемого файла. В этом случае message_data указывает на структуру YR_MODULE_IMPORT. Эта структура содержит поле module_name, указывающее на строку с нулевым символом в конце с именем импортируемого модуля, и два других поля: module_data и module_data_size. Эти поля изначально установлены в NULL и 0, но ваша программа может назначить указатель на некоторые произвольные данные для module_data при установке в module_data_size размера этих данных. Таким образом, вы можете передавать дополнительные данные тем модулям, которым это необходимо, например, модулю Cuckoo.

Как только модуль импортирован, обратный вызов вызывается с сообщением CALLBACK_MSG_MODULE_IMPORTED. Когда это происходит, message_data указывает на структуру YR_OBJECT_STRUCTURE. Эта структура содержит всю информацию, предоставленную модулем о сканируемом в данный момент файле.

Наконец, функция обратного вызова также вызывается с сообщением CALLBACK_MSG_SCAN_FINISHED, когда сканирование завершено. В этом случае message_data имеет значение NULL.

Функция обратного вызова должна возвращать одно из следующих значений:

```
CALLBACK_CONTINUE
CALLBACK_ABORT
CALLBACK_ERROR
```

Если она возвращает CALLBACK_CONTINUE, YARA продолжит сканирование, CALLBACK_ABORT прервет сканирование, но результатом функции yr_rules_scan_XXXX будет ERROR_SUCCESS. С другой стороны, CALLBACK_ERROR также прервет сканирование, но результат из yr_rules_scan_XXXX будет ERROR_CALLBACK_ERROR.

Аргумент user_data, передаваемый в функцию обратного вызова, является таким же, что и аргумент передаваемый в yr_rules_scan_xxxx. Этот указатель не касается YARA, это просто способ для вашей программы передать произвольные данные в функцию обратного вызова.

Все функции yr_rules_scan_XXXX получают аргумент flags и аргумент timeout. Единственный флаг, определенный в настоящее время, это SCAN_FLAGS_FAST_MODE, поэтому вы должны передать либо этот флаг, либо нулевое значение. Аргумент timeout заставляет функцию возвращаться через указанное количество секунд, а ноль означает отсутствие тайм-аута вообще.

Флаг SCAN_FLAGS_FAST_MODE делает сканирование немного быстрее, избегая многократного совпадения одной и той же строки, когда в этом нет необходимости. Как только строка найдена в файле, она впоследствии игнорируется, подразумевая, что у вас будет одно совпадение для строки, даже если она появляется несколько раз в отсканированных данных. Этот флаг имеет тот же эффект, что и параметр командной строки -f, описанный в Главе 5 Запуск YARA из командной строки.

Обратите внимание, что вы не должны вызвать любую из функций yr_rules_scan_XXXX из функции обратного вызова, так как эти функции не реентерабельны.

Использование сканера

Функции yr_rules_scan_XXXX достаточно в большинстве случаев, но иногда может понадобиться более детальный контроль над сканированием. В этих случаях вы можете создать сканер с помощью функции yr_scanner_create(). Сканер - это просто оболочка вокруг структуры YR_RULES, которая содержит дополнительную конфигурацию, например внешние переменные, не затрагивая других пользователей структуры YR_RULES.

Сканер особенно полезен, когда вы хотите использовать один и тот же YR_RULES с несколькими потребителями (это может быть отдельный поток, сопрограмма и т. д.), И каждому потребителю необходимо установить различный набор значений для внешних переменных. В этом случае вы не можете использовать yr_rules_define_XXXX_variable для установки значений ваших внешних переменных, поскольку такие изменения затронут каждого потребителя, использующего YR_RULES. Однако у каждого потребителя может быть свой собственный сканер, в котором сканеры используют один и тот же YR_RULES и используют yr_scanner_define_XXXX_variable для установки внешних переменных без влияния на остальных потребителей.

Это лучшее решение, чем иметь отдельный YR_RULES для каждого потребителя, так как структуры YR_RULES имеют большой объем памяти (особенно если у вас много правил), в то время как сканеры гораздо менее ресурсоемкие.

1.7.6 Описание API

Структуры данных

YR_COMPILER

Структура данных, представляющая компилятор YARA.

YR_MATCH

Структура данных, представляющая строковое соответствие.

- int64_t base - Базовое смещение/адрес совпадения. При сканировании файла это поле обычно равно нулю, а при сканировании пространства памяти процесса это поле является виртуальным адресом блока памяти, в котором было найдено совпадение.
- int64_t offset - Смещение совпадения относительно base.
- int32_t match_length - Длина совпадающей строки.
- const uint8_t* data - Указатель на буфер, содержащий часть совпадающей строки.
- int32_t data_length - Длина буфера data. Минимальное значение data_length-match_length, максимальное - MAX_MATCH_DATA.

Изменено в версии 3.5.0.

YR_META

Структура данных, представляющая значения метаданных.

- const char* identifier - Идентификатор метаданных.
- int32_t type - Один из следующих типов метаданных:
  - META_TYPE_NULL
  - META_TYPE_INTEGER
  - META_TYPE_STRING
  - META_TYPE_BOOLEAN

YR_MODULE_IMPORT

- const char* module_name - Имя импортируемого модуля.
- void* module_data - Указатель на дополнительные данные, передаваемые в модуль. Первоначально установленный в NULL, при этом ваша программа отвечает за установку этого указателя при обработке сообщения CALLBACK_MSG_IMPORT_MODULE.
- size_t module_data_size - Размер дополнительных данных, передаваемых в модуль. Ваша программа должна установить соответствующее значение, если module_data изменен.

YR_RULE

Структура данных, представляющая одно правило.

- const char* identifier - Идентификатор правила.
- const char* tags - Указатель на последовательность строк с нулевым символом в конце с именами тегов. Дополнительный нулевой символ отмечает конец последовательности. Пример: tag1\0tag2\0tag3\0\0. Для перебора тегов вы можете использовать yr_rule_tags_foreach().
- YR_META* metas - Указатель на последовательность структур YR_META. Для перебора структур используйте yr_rule_metas_foreach().
- YR_STRING* strings - Указатель на последовательность структур YR_STRING. Для перебора структур используйте yr_rule_strings_foreach().
- YR_NAMESPACE* ns - Указатель на структуру YR_NAMESPACE.

YR_RULES

Структура данных, представляющая набор правил.

YR_STREAM

Добавлено в версии 3.4.0. Структура данных, представляющая поток, используемый с функциями yr_rules_load_stream() и yr_rules_save_stream().

- void* user_data - Пользовательский указатель.

- YR_STREAM_READ_FUNC read - Указатель на функцию потока ‘read’, предоставленную пользователем.
- YR_STREAM_WRITE_FUNC write - Указатель на функцию потока ‘write’, предоставленную пользователем.

YR_STRING

Структура данных, представляющая строку, объявленную в правиле.

- const char* identifier - Идентификатор строки.

YR_NAMESPACE

Структура данных, представляющая пространство имен правила.

- const char* name - Пространство имен правила.

Функции

int yr_initialize (void)

Инициализация библиотеки. Должна быть вызвана главным потоком перед использованием любой другой функции. Возвращает ERROR_SUCCESS в случае успеха, либо другой код ошибки в случае неудачи. Список возможных кодов возврата варьируется в зависимости от модулей, скомпилированных в YARA.

---

int yr_finalize (void)

Завершает работу библиотеки. Должна вызываться основным потоком для освобождения любого ресурса, выделенного библиотекой. Возвращает ERROR_SUCCESS в случае успеха, либо другой код ошибки в случае неудачи. Список возможных кодов возврата зависит от модулей, скомпилированных в YARA.

---

void yr_finalize_thread (void)

Устаревшая начиная с версии 3.8.0 функция. Любой поток, использующий библиотеку, кроме основного потока, должен вызывать эту функцию, при завершении использования библиотеки. Начиная с версии 3.8.0, вызов этой функции больше не требуется.

---

int yr_compiler_create (YR_COMPILER** compiler)

Создает компилятор YARA. В качестве параметра передается адрес указателя на YR_COMPILER, при этом функция установит указатель на вновь выделенный компилятор. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_MEMORY

---

void yr_compiler_destroy (YR_COMPILER* compiler)

Уничтожает компилятор YARA.

---

void yr_compiler_set_callback (YR_COMPILER* compiler, YR_COMPILER_CALLBACK_FUNC callback, void* user_data)

Изменено начиная с версии 3.3.0. Устанавливает обратный вызов для получения информации об ошибке и предупреждении. Указатель на user_data передается в функцию обратного вызова.

---

void yr_compiler_set_include_callback (YR_COMPILER* compiler, YR_COMPILER_INCLUDE_CALLBACK_FUNC callback, YR_COMPILER_INCLUDE_FREE_FUNC include_ free, void* user_data)

Устанавливает обратный вызов для предоставления правил из пользовательского источника при вызове директивы include. Указатель user_data остается нетронутым и передается назад в функцию обратного вызова и в свободную функцию. Как только результат обратного вызова больше не нужен, будет вызвана функция include_free. Если память не должна быть освобождена, include_free может быть присвоено значение null. Если обратный вызов имеет значение NULL, поддержка директив include отключена.

---

int yr_compiler_add_file (YR_COMPILER* compiler, FILE* file, const char* namespace, const char* file_name)

Компилирует правила из файла file. Правила помещаются в пространство имен namespace, если namespace равно NULL, они будут помещены в пространство имен по умолчанию. file_name - это имя файла для создания отчетов об ошибках, которое может иметь значение NULL. Возвращает количество ошибок, обнаруженных во время компиляции.

---

int yr_compiler_add_fd (YR_COMPILER* compiler, YR_FILE_DESCRIPTOR rules_fd, const char* namespace, const char* file_name)

Добавлено в версии 3.6.0. Компилирует правила из файлового дескриптора rules_fd. Правила помещаются в пространство имен namespace, если namespace равно NULL, они будут помещены в пространство имен по умолчанию. file_name - это имя файла для создания отчетов об ошибках, которое может иметь значение NULL. Возвращает количество ошибок, обнаруженных во время компиляции.

---

int yr_compiler_add_string (YR_COMPILER* compiler, const char* string, const char* namespace)

Компилирует правила из строки string. Правила помещаются в пространство имен namespace, если namespace равно NULL, они будут помещены в пространство имен по умолчанию. file_name - это имя файла для создания отчетов об ошибках, которое может иметь значение NULL. Возвращает количество ошибок, обнаруженных во время компиляции.

---

int yr_compiler_get_rules (YR_COMPILER* compiler, YR_RULES** rules)

Получает скомпилированные правила из компилятора. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS

- ERROR_INSUFFICIENT_MEMORY

---

int yr_compiler_define_integer_variable (YR_COMPILER* compiler, const char* identifier, int64_t value)

Определяет внешнюю целочисленную переменную.

---

int yr_compiler_define_float_variable (YR_COMPILER* compiler, const char* identifier, double value)

Определяет внешнюю переменную с плавающей точкой.

---

int yr_compiler_define_boolean_variable (YR_COMPILER* compiler, const char* identifier, int value)

Определяет внешнюю переменную типа boolean.

---

int yr_compiler_define_string_variable (YR_COMPILER* compiler, const char* identifier, const char* value)

Определяет внешнюю строковую переменную.

---

int yr_rules_define_integer_variable (YR_RULES* rules, const char* identifier, int64_t value)

Определяет внешнюю целочисленную переменную.

---

int yr_rules_define_boolean_variable (YR_RULES* rules, const char* identifier, int value)

Определяет внешнюю переменную типа boolean.

---

int yr_rules_define_float_variable (YR_RULES* rules, const char* identifier, double value)

Определяет внешнюю переменную с плавающей точкой.

---

int yr_rules_define_string_variable (YR_RULES* rules, const char* identifier, const char* value)

Определяет внешнюю строковую переменную.

---

void yr_rules_destroy (YR_RULES* rules)

Уничтожает скомпилированные правила .

---

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

Сохраняет скомпилированные правила в файл, указанный в filename. Можно сохранить только правила, полученные с помощью yr_compiler_get_rules(). Правила, полученные с помощью yr_rules_load() или yr_rules_load_stream() не могут быть сохранены. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS

- ERROR_COULD_NOT_OPEN_FILE

---

int yr_rules_save_stream (YR_RULES* rules, YR_STREAM* stream)

Добавлено в версии 3.4.0.

Сохраняет скомпилированные правила rules в stream. Можно сохранить только правила, полученные с помощью yr_compiler_get_rules(). Правила, полученные с помощью yr_rules_load() или yr_rules_load_stream() не могут быть сохранены. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS

---

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

Загружает скомпилированные правила rules из файла, указанного в параметре filename. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_MEMORY
- ERROR_COULD_NOT_OPEN_FILE
- ERROR_INVALID_FILE
- ERROR_CORRUPT_FILE
- ERROR_UNSUPPORTED_FILE_VERSION

---

int yr_rules_load_stream (YR_STREAM* stream, YR_RULES** rules) Добавлено в версии 3.4.0.

Загружает скомпилированные правила rules из потока stream. Правила, загруженные таким образом, не могут быть сохранены обратно с помощью yr_rules_save_stream(). Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_MEMORY
- ERROR_INVALID_FILE
- ERROR_CORRUPT_FILE
- ERROR_UNSUPPORTED_FILE_VERSION

---

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

Сканирование участка памяти buffer. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_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)

Сканирование файла. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_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

---

int yr_rules_scan_fd (YR_RULES* rules, YR_FILE_DESCRIPTOR fd, int flags, YR_CALLBACK_FUNC callback, void* user_data, int timeout)

Сканирование файла по его дескриптору . В системах POSIX YR_FILE_DESCRIPTOR - это int, возвращаемый функцией open (). В Windows YR_FILE_DESCRIPTOR - это HANDLE, возвращаемый CreateFile (). Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_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)

Повторение по тегам данного правила, выполняя блок кода, который следует каждый раз, с другим значением для tag типа const char *. Например:

```
const char* tag;

/* rule - объект YR_RULE */

r_rule_tags_foreach(rule, tag)
{
      ..do //сделать что-нибудь с tag
}
```

---

yr_rule_metas_foreach (rule, meta)

Выполняет повторение по структуре YR_META*, связанной с данным правилом, в котором выполняется блок кода, который каждый раз следует с другим значением для meta. Например:

```
YR_META* meta;

/* rule - объект YR_RULE */

yr_rule_metas_foreach(rule, meta)
{
      ..do //сделать что-нибудь с meta
}
```

---

yr_rule_strings_foreach (rule, string)

Выполняет повторение по структуре YR_STRING*, связанной с данным правилом, в котором выполняется блок кода, который каждый раз следует с другим значением для string. Например:

```
YR_STRING* string;

/* rule - объект YR_RULE */

yr_rule_strings_foreach(rule, string)
{
      ..do //сделать что-нибудь с string
}
```

---

yr_string_matches_foreach (string, match)

Выполняет повторение по структуре YR_MATCH*, связанной с данным правилом, в котором выполняется блок кода, который каждый раз следует с другим значением для match. Например:

```
YR_MATCH* match;

/* string - объект YR_STRING */

yr_string_matches_foreach(string, match)
{
      ..do //сделать что-нибудь с match
}
```

---

yr_rules_foreach (rules, rule)

Повторение по каждому YR_RULE в объекте YR_RULES, выполняя блок кода, который следует каждый раз с другим значением rule. Например:

```
YR_RULE* rule;

/* rule - объект YR_RULE */
```

(continues on next page)

(continued from previous page)

```
yr_rules_foreach(rules, rule)
{
      ..do //сделать что-нибудь с rule
}
```

---

void yr_rule_disable (YR_RULE* rule)

Добавлено в версии 3.7.0.

Отключает указанное правило. Отключенные правила полностью игнорируются в процессе сканирования и не вызывают совпадений. Если отключенное правило используется в для определения состояния какого-либо другого правила, значение для отключенного правила не определено (т. е. не является ни истинным, ни ложным). Дополнительные сведения о неопределенных значениях см. в разделе 2.6.

---

void yr_rule_enable (YR_RULE* rule)

Добавлено в версии 3.7.0.

Включает указанное правило. После использования yr_rule_disable() правило можно заново включить с помощью этой функции.

int yr_scanner_create (YR_RULES* rules, YR_SCANNER **scanner)

Добавлено в версии 3.8.0.

Создает новый сканер, который можно использовать для сканирования данных с помощью предоставленных правил. scanner должен быть указателем на YR_SCANNER, при этом функция установит указатель на вновь выделенный сканер. Возвращает один из следующих кодов ошибок:

- ERROR_INSUFFICIENT_MEMORY

---

void yr_scanner_destroy (YR_SCANNER *scanner)

Добавлено в версии 3.8.0.

Уничтожает сканер. После использования сканера он должен быть уничтожен с помощью этой функции.

---

void yr_scanner_set_callback (YR_SCANNER *scanner, YR_CALLBACK_FUNC callback, void* user_data)

Добавлено в версии 3.8.0.

Устанавливает функцию обратного вызова, которая будет вызываться для сообщения о любых совпадениях, найденных сканером.

---

void yr_scanner_set_timeout (YR_SCANNER* scanner, int timeout)

Добавлено в версии 3.8.0.

Устанавливает максимальное количество секунд, которое сканер будет тратить при любом вызове yr_scanner_scan_xxx.

---

void yr_scanner_set_flags (YR_SCANNER* scanner, int flags)

Добавлено в версии 3.8.0.

Устанавливает флаги, которые будут использоваться при любом вызове yr_scanner_scan_xxx.

---

int yr_scanner_define_integer_variable (YR_SCANNER* scanner, const char* identifier, int64_t value)

Добавлено в версии 3.8.0.

Определяет внешнюю целочисленную переменную.

---

int yr_scanner_define_boolean_variable (YR_SCANNER* scanner, const char* identifier, int value)

Добавлено в версии 3.8.0.

Определяет внешнюю переменную типа boolean.

---

int yr_scanner_define_float_variable (YR_SCANNER* scanner, const char* identifier, double value)

Добавлено в версии 3.8.0.

Определяет внешнюю переменную с плавающей точкой.

---

int yr_scanner_define_string_variable (YR_SCANNER* scanner, const char* identifier, const char* value)

Добавлено в версии 3.8.0.

Определяет внешнюю строковую переменную.

---

int yr_scanner_scan_mem (YR_SCANNER* scanner, const uint8_t* buffer, size_t buffer_size)

Добавлено в версии 3.8.0.

Сканирует область памяти. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_MEMORY
- ERROR_TOO_MANY_SCAN_THREADS
- ERROR_SCAN_TIMEOUT
- ERROR_CALLBACK_ERROR
- ERROR_TOO_MANY_MATCHES

---

int yr_scanner_scan_file (YR_SCANNER* scanner, const char* filename)

Добавлено в версии 3.8.0.

Сканирует файл. Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_MEMORY

- ERROR_TOO_MANY_SCAN_THREADS
- ERROR_SCAN_TIMEOUT
- ERROR_CALLBACK_ERROR
- ERROR_TOO_MANY_MATCHES

---

int yr_scanner_scan_fd (YR_SCANNER* scanner, YR_FILE_DESCRIPTOR fd)

Добавлено в версии 3.8.0.

Сканирование файла по его дескриптору. В системах POSIX YR_FILE_DESCRIPTOR - это int, возвращаемый функцией open(). В Windows YR_FILE_DESCRIPTOR - это дескриптор, возвращаемый функцией CreateFile(). Возвращает один из следующих кодов ошибок:

- ERROR_SUCCESS
- ERROR_INSUFFICIENT_MEMORY
- ERROR_TOO_MANY_SCAN_THREADS
- ERROR_SCAN_TIMEOUT
- ERROR_CALLBACK_ERROR
- ERROR_TOO_MANY_MATCHES

Коды ошибок

ERROR_SUCCESS

Все прошло нормально.

ERROR_INSUFFICIENT_MEMORY

Недостаточно памяти для завершения операции.

ERROR_COULD_NOT_OPEN_FILE

Файл не может быть открыт.

ERROR_COULD_NOT_MAP_FILE

Файл не может быть отображен в память.

ERROR_ZERO_LENGTH_FILE

Длина файла равна нулю.

ERROR_INVALID_FILE

Файл не является допустимым файлом правил.

ERROR_CORRUPT_FILE

Файл правил поврежден.

ERROR_UNSUPPORTED_FILE_VERSION

Файл сгенерирован другой версией YARA и не может быть загружен этой версией.

ERROR_TOO_MANY_SCAN_THREADS

Слишком много потоков пытаются использовать один и тот же объект YR_RULES одновременно. Предел определяется YR_MAX_THREADS в ./include/yara/limits.h.

ERROR_SCAN_TIMEOUT

Время сканирования истекло.

ERROR_CALLBACK_ERROR

Функция обратного вызова вернула ошибку.

ERROR_TOO_MANY_MATCHES

Слишком много совпадений для какой-либо строки в правилах. Обычно это происходит, когда правила содержат очень короткие или очень распространенные строки, такие как 01 02 или FF FF FF FF. Предел определяется YR_MAX_STRING_MATCHES в ./include/yara/limits.h.