EQL Documentation

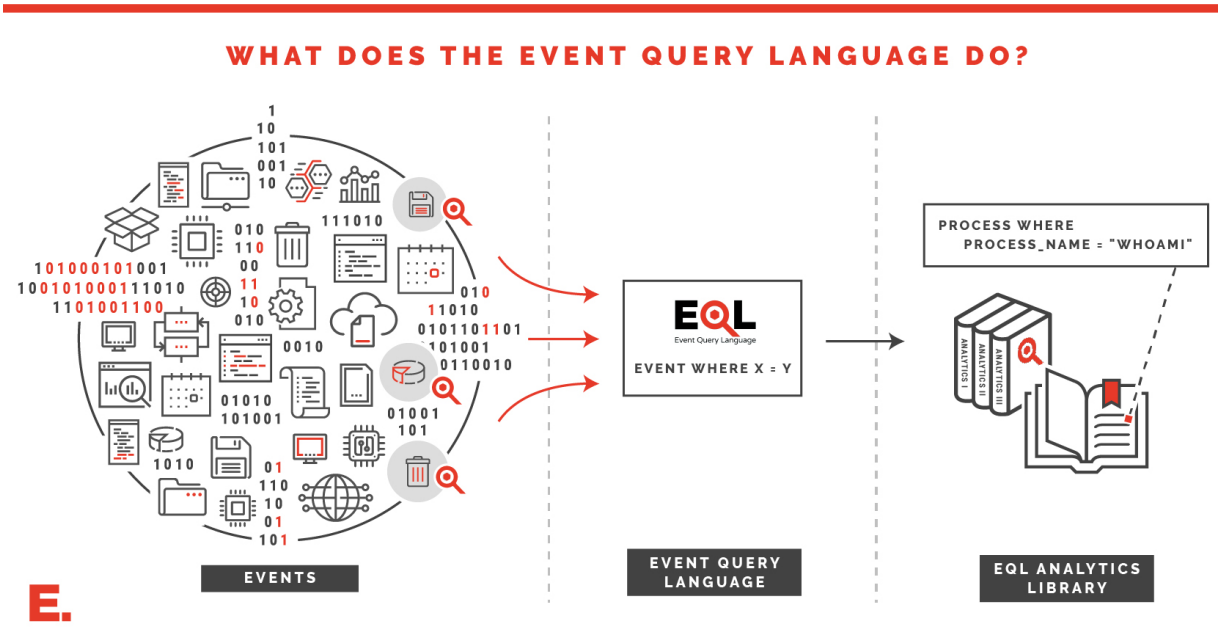
Release 0.8.4

Endgame

Mar 17, 2020

Contents

EQL is a language that can match events, generate sequences, stack data, build aggregations, and perform analysis. EQL is schemaless and supports multiple database backends. It supports field lookups, boolean logic, comparisons, wildcard matching, and function calls. EQL also has a preprocessor that can perform parse and translation time evaluation, allowing for easily sharable components between queries.



CHAPTER 1

Getting Started

The EQL module current supports Python 2.7 and 3.5+. Assuming a supported Python version is installed, run the command:

```
$ pip install eql
```

If Python is configured and already in the PATH, then `eql` will be readily available, and can be checked by running the command:

```
$ eql --version
eql 0.8
```

From there, try a `sample json file` and test it with EQL.

```
$ eql query -f example.json "process where process_name == 'explorer.exe'"

{"command_line": "C:\\Windows\\Explorer.EXE", "event_type": "process", "md5":
↪"ac4c51eb24aa95b77f705ab159189e24", "pid": 2460, "ppid": 3052, "process_name":
↪"explorer.exe", "process_path": "C:\\Windows\\explorer.exe", "subtype": "create",
↪"timestamp": 131485997150000000, "user": "research\\researcher", "user_domain":
↪"research", "user_name": "researcher"}
```

CHAPTER 2

Next Steps

- Check out the *Query Guide* for a crash course on writing EQL queries
- View usage for the *Interactive Shell*
- Explore the *API Reference* for advanced usage or incorporating EQL into other projects
- Browse a library of EQL analytics

2.1 Query Guide

2.1.1 Basic Syntax

Basic queries within EQL require an event type and a matching condition. The two are connected using the `where` keyword.

At the most basic level, an event query has the structure:

```
event where condition
```

More specifically, an event query may resemble:

```
process where process_name == "svchost.exe" and command_line != "* -k *"
```

Conditions

Individual events can be matched with EQL by specifying criteria to match the fields in the event to other fields or values. Criteria can be combined with

Boolean operators

```
and  or  not
```

Value comparisons

```
<  <=  ==  !=  >=  >
```

Mathematical operations New in version 0.8.

```
+  -  *  /  %
```

Wildcard matching

```
name == "*some*glob*match*"
name != "*some*glob*match*"
```

Function calls

```
concat(user_domain, "\\", user_name)
length(command_line) > 400
add(timestamp, 300)
```

Method syntax for concise function calls

```
command_line:length() > 400
```

Lookups against static or dynamic values New in version 0.8: Support for `not in`

```
user_name in ("Administrator", "SYSTEM", "NETWORK SERVICE")
user_name not in ("Administrator", "SYSTEM", "NETWORK SERVICE")
process_name in ("cmd.exe", parent_process_name)
```

Strings

Strings are represented with single quotes `'` or double quotes `"`, with special characters escaped by a single backslash. Additionally, raw strings are represented with a leading `?` character before the string, which disables escape sequences for all characters except the quote character.

```
"hello world"
"hello world with 'substring'"
'example \t of \n escaped \r characters'
?"String with literal 'slash' \ characters included"
```

Event Relationships

Relationships between events can be used for stateful tracking within the query. If a related event exists that matches the criteria, then it is evaluated in the query as `true`. Relationships can be arbitrarily nested, allowing for complex behavior and state to be tracked. Existing relationships include `child of`, `descendant of` and `event of`.

Network activity for PowerShell processes that were not spawned from explorer.exe

```
network where process_name == "powershell.exe" and
    not descendant of [process where process_name == "explorer.exe"]
```

Grandchildren of the WMI Provider Service

```
process where child of [process where parent_process_name == "wmiprvse.exe"]
```

Text file modifications by command shells with redirection

```
file where file_name == "*.txt" and
    event of [process where process_name == "cmd.exe" and command_line == "* > *"]
```

Executable file modifications by children of PowerShell

```
file where file_name == "*.exe" and event of [
  process where child of [process where process_name == "powershell.exe"]
]
```

2.1.2 Sequences

Many behaviors are more complex and are best described with an ordered `sequence` of multiple events over a short interval. Complex behaviors may share properties between events in the sequence or require careful handling of state.

Core sequence template

```
sequence
  [event_type1 where condition1]
  [event_type2 where condition2]
  ...
  [event_typeN where conditionN]
```

An example of simple behavior that can spans multiple events is a network logon over Remote Desktop. With a `maxspan` of 30 seconds, we would expect to see an incoming network connection from a host, followed by a separate event for the remote authentication success or failure.

```
sequence with maxspan=30s
  [network where destination_port==3389 and event_subtype_full="*_accept_event*"]
  [security where event_id in (4624, 4625) and logon_type == 10]
```

Although the `sequence` connects the two events temporally, it doesn't prove that they are related. There could be incoming attempts over Remote Desktop from multiple computers, leading to more network and security events. The sequence can be constrained `by` matching fields, so that the network connection and the logon event must share the same source host.

```
sequence with maxspan=30s
  [network where destination_port==3389 and event_subtype_full="*_accept_event"] by␣
↪source_address
  [security where event_id in (4624, 4625) and logon_type == 10] by ip_address
```

For some sequences, multiple values need to be shared across the sequence. One example for this is a user that creates a file and shortly executes it.

```
sequence with maxspan=5m
  [ file where file_name == "*.exe"] by user_name, file_path
  [ process where true] by user_name, process_path
```

Since some fields are in common across all events, this could be represented more succinctly by moving `by user_name` to the top of the query.

```
sequence by user_name with maxspan=5m
  [ file where file_name == "*.exe"] by file_path
  [ process where true] by process_path
```

Managing State

Occasionally, a `sequence` needs to carefully manage and expire state. Sequences are valid `until` a specific event occurs. This can help expire non-unique identifiers and reduce memory usage.

Handles and process identifiers are frequently reused. Stateful `sequence` tracking avoids invalid pairs of events. Within Windows, a process identifier (PID) is only unique while a process is running, but can be reused after its termination. When building a `sequence` of process identifiers, a process termination will cause all state to be invalidated and thrown away.

For instance, if `whoami.exe` executed from a batch file, matching ppid of `whoami.exe` to the pid of `cmd.exe` can only be done while the parent process is alive. As a result, the sequence is valid `until` the matching termination event occurs.

```
sequence
  [ process where process_name == "cmd.exe" and command_line == "* *.bat*" and event_
↪subtype_full == "creation_event"] by pid
  [ process where process_name == "whoami.exe" and event_subtype_full == "creation_
↪event"] by ppid
until [ process where event_subtype_full == "termination_event"] by pid
```

2.1.3 Joins

In EQL, `join` is used to link unordered events that may share properties. This is similar to `sequence`, but lacks time constraints.

Basic structure

```
join  // by shared_field1, shared_field2, ...
  [event_type1 where condition1] // by field1
  [event_type2 where condition2] // by field2
  ...
  [event_typeN where conditionN] // by field3
```

This is useful when identifying multiple connections between two network endpoints with different ports. With `join`, events can happen in any order, and when all events match, the `join` is completed.

```
join by source_ip, destination_ip
  [network where destination_port == 3389]  // RDP
  [network where destination_port == 135]   // RPC
  [network where destination_port == 445]   // SMB
```

Like sequences, events can also be joined `until` an expiration event is met. For instance, it may be useful to identify processes with registry, network, and file activity.

```
join by pid
  [process where true]
  [network where true]
  [registry where true]
  [file where true]

until [process where event_subtype_full == "termination_event"]
```

2.1.4 Pipes

Queries can include pipes for post-processing of events, and can be used for enrichment, aggregations, statistics and filtering.

count

The `count` pipe will return only statistics. If no arguments are passed, then it returns the total number of events. Otherwise, it returns the number of occurrences for each unique value. Stats are returned in the form

Count the total number of events

```
process where true | count

// results look like
// {"count": 100, "key": totals"}
```

Count the number of times each value occurs

```
process where true | count process_name

// results look like
// {"count": 100, "key": "cmd.exe", "percent": 0.5}
// {"count": 50, "key": "powershell.exe", "percent": 0.25}
// {"count": 50, "key": "net.exe", "percent": 0.25}
```

Count the number of times a set of values occur

```
process where true | count parent_process_name, process_name

// results look like
// {"count": 100, "key": ["explorer.exe", "cmd.exe"], "percent": 0.5}
// {"count": 50, "key": ["explorer.exe", "powershell.exe"], "percent": 0.25}
// {"count": 50, "key": ["cmd.exe", "net.exe"], "percent": 0.25}
```

unique

The `unique` pipe will only return the first matching result through the pipe. Unless a *sort* pipe exists before it, events will be ordered chronologically.

Get the first matching process for each unique name

```
process where true | unique process_name
```

Get the first result for multiple of values

```
process where true | unique process_name, command_line
```

filter

The `filter` pipe will only output events that match the criteria. With simple queries, this can be accomplished by adding `and` to the search criteria. It's most commonly used to filter sequences or with other pipes.

Find network destinations that were first seen after May 5, 2018

```
network where true
| unique destination_address, destination_port
| filter timestamp_utc >= "2018-05-01"
```

unique_count

The `unique_count` pipe combines the filtering of *unique* with the stats from *count*. For `unique_count`, the original event is returned but with the fields `count` and `percent` added.

Get the first result per unique value(s), with added count information

```
process where true | unique_count process_name | filter count < 5
```

head

The `head` pipe is similar to the UNIX head command and will output the first N events coming through the pipe.

Get the first fifty unique powershell commands

```
process where process_name == "powershell.exe"
| unique command_line
| head 50
```

tail

The `tail` pipe is similar to the UNIX tail command and will output the latest events coming through the pipe.

Get the most recent ten logon events

```
security where event_id == 4624
| tail 10
```

sort

The `sort` pipe will reorder events coming through the pipe. Sorting can be done with one or multiple values.

Warning: In general, `sort` will buffer all events coming into the pipe, and will sort them all at once. It's often good practice to bound the number of events into the pipe.

For instance, the following query could be slow and require significant memory usage on a busy system.

```
file where true | sort file_name
```

Get the top five network connections that transmitted the most data

```
network where total_out_bytes > 100000000
| sort total_out_bytes
| tail 5
```

2.1.5 Functions

Function calls keep the core language for EQL simple but easily extendable. Functions are used to perform math, string manipulation or more sophisticated expressions to be expressed.

add (*x*, *y*)
Returns `x + y`

Changed in version 0.8: Added + operator directly.

arrayContains (*some_array*, *value*[, ...])
Check if `value` is a member of the array `some_array`.

Changed in version 0.7: Support for additional arguments.

```
// {my_array: ["value1", "value2", "value3"]}

arrayContains(my_array, "value2")           // returns true
arrayContains(my_array, "value4")           // returns false
arrayContains(my_array, "value3", "value4)  // returns true
```

arrayCount (*array*, *variable*, *expression*)
Count the number of matches in an array to an expression.

New in version 0.7.

```
// {my_array: [{user: "root", props: [{level: 1}, {level: 2}]},
//             {user: "guest", props: [{level: 1}]}]

arrayCount(my_array, item, item.user == "root")                              //␣
↪returns 1
arrayCount(my_array, item, item.props[0].level == 1)                         //␣
↪returns 2
arrayCount(my_array, item, item.props[1].level == 4)                         //␣
↪returns 0
arrayCount(my_array, item, arrayCount(item.props, p, p.level == 2) == 1)   //␣
↪returns 1
```

arraySearch (*array*, *variable*, *expression*)
Check if any member in the array matches an expression. Unlike *arrayContains()*, this can search over nested structures in arrays, and supports searching over arrays within arrays.

```
// {my_array: [{user: "root", props: [{level: 1}, {level: 2}]},
//             {user: "guest", props: [{level: 1}]}]

arraySearch(my_array, item, item.user == "root")                          // returns␣
↪true
arraySearch(my_array, item, item.props[0].level == 1)                     // returns␣
↪true
arraySearch(my_array, item, item.props[1].level == 4)                     // returns␣
↪false
arraySearch(my_array, item, arraySearch(item.props, p, p.level == 2))   // returns␣
↪true
```

between (*source*, *left*, *right*[, *greedy=false*, *case_sensitive=false*])
Extracts a substring from `source` that's also between `left` and `right`.

Parameters

- **greedy** – Matches the longest string when set, similar to `.*` vs `.*?`.

- **case_sensitive** – Match case when searching for `left` and `right`.

```
between("welcome to event query language", " ", " ")            // returns "to"
between("welcome to event query language", " ", " ", true)      // returns "to␣
↪event query"
```

cidrMatch (*ip_address*, *cidr_block*[, ...])

Returns `true` if the source address matches any of the provided CIDR blocks.

Changed in version 0.8.

```
// ip_address = "192.168.152.12"
cidrMatch(ip_address, "10.0.0.0/8", "192.168.0.0/16")     // returns true
```

concat (...)

Returns a concatenated string of all the input arguments.

```
concat("Process ", process_name, " executed with pid ", pid)
```

divide (*m*, *n*)

Return `m / n`

Changed in version 0.8: Added `/` operator directly.

endsWith (*x*, *y*)

Checks if the string `x` ends with the substring `y`.

indexOf (*source*, *substring*[, *start=0*])

Find the first position (zero-indexed) of a string where a substring is found. If `start` is provided, then this will find the first occurrence at or after the start position.

```
indexOf("some-subdomain.another-subdomain.com", ".")     // returns 14
indexOf("some-subdomain.another-subdomain.com", ".", 14) // returns 14
indexOf("some-subdomain.another-subdomain.com", ".", 15) // returns 32
```

length (*s*)

Returns the length of a string. Non-string values return 0.

match (*source*, *pattern*[, ...])

Checks if multiple regular expressions are matched against a source string.

```
match("event query language", ?"[a-z]+ [a-z]+ [a-z]")   // returns true
```

modulo (*m*, *n*)

Performs the modulo operator and returns the remainder of `m / n`.

Changed in version 0.8: Added `%` operator directly.

multiply (*x*, *y*)

Returns `x * y`

Changed in version 0.8: Added `*` operator directly.

number (*s*[, *base=10*])

Parameters **base** (*number*) – The base of a number.

Returns a number constructed from the string `s`.

```
number("1337")                  // returns 1337
number("0xdeadbeef", 16)        // 3735928559
```

startsWith (*x*, *y*)
Checks if the string `x` starts with the string `y`.

string (*val*)
Returns the string representation of the value `val`.

stringContains (*a*, *b*)
Returns true if `b` is a substring of `a`

substring (*source*[, *start*, *end*])
Extracts a substring between from another string between `start` and `end`. Like other EQL functions, `start` and `end` are zero-indexed positions in the string. Behavior is similar to Python's string slicing (`source[start:end]`), and negative offsets are supported.

```
substring("event query language", 0, 5)                    // returns "event"
substring("event query language", 0, length("event"))      // returns "event"
substring("event query language", 6, 11)                   // returns "query"
substring("event query language", -8)                      // returns "language"
substring("event query language", -length("language"))     // returns "language"
substring("event query language", -5, -1))                 // returns "guag"
```

subtract (*x*, *y*)
Returns `x - y`

wildcard (*value*, *wildcard*[, ...])
Compare a value to a list of wildcards. Returns true if any of them match. For example, the following two expressions are equivalent.

```
command_line == "* create *" or command_line == "* config *" or command_line ==
↪"* start *"

wildcard(command_line, "* create *", "* config *", "* start *")
```

Methods

Calling functions with values returned from other functions can often be difficult to read for complex expressions. EQL also provides an alternative method syntax that flows more naturally from left to right.

For instance, the expression:

```
length(between(command_line, "-enc ", " ")) > 500
```

is equivalent to the method syntax:

```
command_line:between(command_line, "-enc ", " "):length() > 500
```

2.1.6 Implementation Details

There are optimizations for `sequence` and `join` that eliminate excessive pairing of events and enable efficient processing of a stream of events. This is different from common database relationships, such as a SQL Join, which matches every possible pairing and can potentially be costly for event analytics.

Sequences

The underlying structure of a `sequence` roughly resembles a state machine of events, meaning that only one pending sequence can be in a node at a time. If the sequence uses `by` for matching fields, then multiple pending sequences can exist in a given node as long as values the values matched with `by` are distinct. When a pending sequence matches an event, it will override any pending sequences in the next state with identical `by` values.

The state changes are described for the per-user `sequence` and enumeration events below.

```
sequence by user_name
  [process where process_name == "whoami"]
  [process where process_name == "hostname"]
  [process where process_name == "ifconfig"]
```

```
{id:  1, event_type: "process", user_name: "root", process_name: "whoami"}
{id:  2, event_type: "process", user_name: "root", process_name: "whoami"}
{id:  3, event_type: "process", user_name: "user", process_name: "hostname"}
{id:  4, event_type: "process", user_name: "root", process_name: "hostname"}
{id:  5, event_type: "process", user_name: "root", process_name: "hostname"}
{id:  6, event_type: "process", user_name: "user", process_name: "whoami"}
{id:  7, event_type: "process", user_name: "root", process_name: "whoami"}
{id:  8, event_type: "process", user_name: "user", process_name: "hostname"}
{id:  9, event_type: "process", user_name: "root", process_name: "ifconfig"}
{id: 10, event_type: "process", user_name: "user", process_name: "ifconfig"}
{id: 11, event_type: "process", user_name: "root", process_name: "ifconfig"}
```

Since the sequence is separated `by user_name`, commands executed by `root` and `user` are independently sequenced.

```
{id:  1, event_type: "process", user_name: "root", process_name: "whoami"}
// sequence [1] created in root's state 1

{id:  2, event_type: "process", user_name: "root", process_name: "whoami"}
// sequence [2] overwrote root's state 1

{id:  3, event_type: "process", user_name: "user", process_name: "hostname"}
// nothing happens, because user has an empty state 1

{id:  4, event_type: "process", user_name: "root", process_name: "hostname"}
// sequence [2, 4] now in root's state 2
// root's state 1 is empty

{id:  5, event_type: "process", user_name: "root", process_name: "hostname"}
// root's state 1 is empty, so nothing happens

{id:  6, event_type: "process", user_name: "user", process_name: "whoami"}
// sequence [6] created in user's state 1

{id:  7, event_type: "process", user_name: "root", process_name: "whoami"}
// sequence [7] created in root's state 1

{id:  8, event_type: "process", user_name: "user", process_name: "hostname"}
// sequence [6, 8] now in user's state 2
// user's state 1 is now empty

{id:  9, event_type: "process", user_name: "root", process_name: "ifconfig"}
// sequence [2, 4, 9] completes the sequence for root
```

(continues on next page)

(continued from previous page)

```
// root still has [6] in state 1

{id: 10, event_type: "process", user_name: "user", process_name: "ifconfig"}
// sequence [6, 8, 10] completes the sequence for user

{id: 11, event_type: "process", user_name: "root", process_name: "ifconfig"}
// nothing happens because root has an empty state 2
```

2.1.7 Grammar

An external dependency for EQL is the Python library Lark. Lark generates a parser generator for the below grammar, which EQL uses to parse queries.

```
definitions: definition*
?definition: macro | constant

macro:    "macro" name "(" [name ("," name)*] ")" expr
constant: "const" name EQUALS literal

query_with_definitions: definitions piped_query
piped_query: base_query [pipes]
           | pipes
base_query: sequence
          | join
          | event_query
event_query: [name "where"] expr
sequence: "sequence" [join_values with_params? | with_params join_values?] subquery_
↪by subquery_by+ [until_subquery_by]
join: "join" join_values? subquery_by subquery_by+ until_subquery_by?
until_subquery_by.2: "until" subquery_by
pipes: pipe+
pipe: "|" name [single_atom single_atom+ | expressions]

join_values.2: "by" expressions
?with_params.2: "with" named_params
kv: name [EQUALS (time_range | atom)]
time_range: number name
named_params: kv ("," kv)*
subquery_by: subquery named_params? join_values?
subquery: "[" event_query "]"


// Expressions
expressions: expr ("," expr)* [","]
?expr: or_expr
?or_expr: and_expr ("or" and_expr)*
?and_expr: not_expr ("and" not_expr)*
?not_expr.3: NOT_OP* term
?term: sum_expr comp_op sum_expr -> comparison
     | sum_expr "not" "in" "(" expressions [","]? ")"  -> not_in_set
     | sum_expr "in" "(" expressions [","]? ")" -> in_set
     | sum_expr

// Need to recover these tokens
EQUALS: "==" | "="
```

(continues on next page)

(continued from previous page)

```
COMP_OP: "<=" | "<" | "!=" | ">=" | ">"
?comp_op: EQUALS | COMP_OP
MULT_OP:    "*" | "/" | "%"
NOT_OP:     "not"

method: ":" name "(" [expressions] ")"


?sum_expr: mul_expr (SIGN mul_expr)*
?mul_expr: named_subquery_test (MULT_OP named_subquery_test)*
?named_subquery_test: named_subquery
                    | method_chain
named_subquery.2: name "of" subquery
?method_chain: value (":" function_call)*
?value: SIGN? function_call
      | SIGN? atom
function_call.2: name "(" [expressions] ")"
?atom: single_atom
     |  "(" expr ")"
?signed_single_atom: SIGN? single_atom
?single_atom: literal
            | field
            | base_field
base_field: name
field: FIELD
literal: number
       | string
number: UNSIGNED_INTEGER
      | DECIMAL
string: DQ_STRING
      | SQ_STRING
      | RAW_DQ_STRING
      | RAW_SQ_STRING

// Check against keyword usage
name: NAME

// Tokens
// make this a token to avoid ambiguity, and make more rigid on whitespace
// sequence by pid [1] [true] looks identical to:
// sequence by pid[1] [true]
FIELD: NAME ("." WHITESPACE* NAME | "[" WHITESPACE* UNSIGNED_INTEGER WHITESPACE* "]")+
LCASE_LETTER: "a".."z"
UCASE_LETTER: "A".."Z"
DIGIT: "0".."9"

LETTER: UCASE_LETTER | LCASE_LETTER
WORD: LETTER+

NAME: ("_"|LETTER) ("_"|LETTER|DIGIT)*
UNSIGNED_INTEGER: /[0-9]+/
EXPONENT: /[Ee][-+]?\d+/
DECIMAL: UNSIGNED_INTEGER? "." UNSIGNED_INTEGER+ EXPONENT?
       | UNSIGNED_INTEGER EXPONENT
SIGN:           "+" | "-"
DQ_STRING:      /"(\\[btnfr"'\\]|[^\r\n"\\])*"/
SQ_STRING:      /'(\\[btnfr"'\\]|[^\r\n'\\])*'/
```

(continues on next page)

(continued from previous page)

```
RAW_DQ_STRING:  /\?"(\\\"|[^"\r\n])*"/
RAW_SQ_STRING:  /\?'(\\\'|[^'\r\n])*'/

%import common.NEWLINE

COMMENT: "//" /[^\n]*/
ML_COMMENT: "/*" /(.|\n|\r)*?/ "*/"
WHITESPACE: (" " | "\r" | "\n" | "\t" )+

%ignore COMMENT
%ignore ML_COMMENT
%ignore WHITESPACE
```

2.2 Interactive Shell

The EQL python package provides an interactive shell for data exploration, as well as commands to directly search over JSON and output matches to the console. First install Python and then use `pip` to install EQL.

```
$ pip install eql
```

For the optimal shell experience, use Python 3.6+ and install the optional dependencies for EQL:

```
$ pip install eql[cli]
```

Once the shell is installed. Run the `eql` command to interact with and search data sets. Type `help` within the shell to get a list of commands and `exit` when finished.

Note: In Python 2.7, the argument parsing is a little different. Instead of running `eql` directly to invoke the interactive shell, run `eql shell`.

In addition, the `query` command within EQL will stream over JSON, and output as matches are found. An input file can be provided with `-f` in JSON or as lines of JSON (`.jsonl`). Lines of JSON can also be processed as streams from stdin.

```
$ eql query 'process where true | head 1' -f input.json
{"timestamp": 131485083040000000, "process_name": "System Idle Process"}

$ eql query "process where true | head 1" < input.jsonl
{"timestamp": 131485083040000000, "process_name": "System Idle Process"}

$ cat input.jsonl | eql query "process where true" | head -n 1
{"timestamp": 131485083040000000, "process_name": "System Idle Process"}

$ eql query "process where true | count process_name | head 3" -f tmp.jsonl
{"count": 1, "percent": 0.125, "key": "application.exe"}
{"count": 2, "percent": 0.25, "key": "software.exe"}
{"count": 2, "percent": 0.25, "key": "tools.exe"}
```

Additionally, the CLI allows for pieces of the query to be missing. The base query `process where true` can be skipped altogether if pipes are present.

```
$ eql query '| head 1' -f input.jsonl
{"timestamp": 131485083040000000, "process_name": "System Idle Process"}
```

Additionally, `any where process_name == "application.exe"` is equivalent to `process_name == "application.exe"`

```
$ eql query "process_name == '*.exe' | count process_name | head 3" -f tmp.jsonl
{"count": 1, "percent": 0.125, "key": "application.exe"}
{"count": 2, "percent": 0.25, "key": "software.exe"}
{"count": 2, "percent": 0.25, "key": "tools.exe"}
```

2.2.1 Detailed Usage

```
$ eql -h
usage: eql [-h] [--version] {build,query} ...
```

eql build

```
$ eql build -h
usage: eql build [-h] [--config CONFIG] [--analytics-only] input_file output_file

positional arguments:
  input_file       Input analytic file(s) (.json, .yml, .toml)
  output_file      Output engine file

optional arguments:
  --config CONFIG  Engine configuration
  --analytics-only     Skips core engine when building target
```

eql query

```
$ eql query -h
usage: eql query [-h] [--file FILE] [--encoding ENCODING]
                 [--format {json,jsonl}] [--config CONFIG]
                 query

positional arguments:
  query                 The EQL query to run over the log file

optional arguments:
  --file FILE, -f FILE  Target file(s) to query with EQL
  --encoding ENCODING, -e ENCODING
                        Encoding of input file (utf8, utf16, etc)
  --format {json,jsonl,json.gz,jsonl.gz}
                        File type. If not specified, defaults to the extension for --
↪file
  --config CONFIG       Engine configuration
```

2.3 API Reference

2.3.1 Parser

eql.**get_preprocessor**(*text*, *implied_any=False*, *subqueries=None*, *preprocessor=None*)
Parse EQL definitions and get a `PreProcessor`.

Parameters

- **text** (*str*) – EQL source to parse
- **preprocessor** (*PreProcessor*) – Use an existing EQL preprocessor while parsing definitions
- **implied_any** (*bool*) – Allow for event queries to match on any event type when a type is not specified. If enabled, the query `process_name == "cmd.exe"` becomes `any where process_name == "cmd.exe"`
- **subqueries** (*bool*) – Toggle support for subqueries, which are required by `descendant of`, `child of` and `event of`

Return type PreProcessor

eql.**parse_definitions**(*text*, *preprocessor=None*, *implied_any=False*, *subqueries=True*)
Parse EQL preprocessor definitions from source.

Parameters

- **text** (*str*) – EQL source to parse
- **preprocessor** (*PreProcessor*) – Use an EQL preprocessor to expand definitions and constants while parsing
- **implied_any** (*bool*) – Allow for event queries to match on any event type when a type is not specified. If enabled, the query `process_name == "cmd.exe"` becomes `any where process_name == "cmd.exe"`
- **subqueries** (*bool*) – Toggle support for subqueries, which are required by `sequence`, `join`, `descendant of`, `child of` and `event of`

Return type list[Definition]

eql.**parse_expression**(*text*, *implied_any=False*, *preprocessor=None*, *subqueries=True*)
Parse an EQL expression and return the AST.

Parameters

- **text** (*str*) – EQL source text to parse
- **implied_any** (*bool*) – Allow for event queries to match on any event type when a type is not specified. If enabled, the query `process_name == "cmd.exe"` becomes `any where process_name == "cmd.exe"`
- **subqueries** (*bool*) – Toggle support for subqueries, which are required by `sequence`, `join`, `descendant of`, `child of` and `event of`
- **preprocessor** (*PreProcessor*) – Optional preprocessor to expand definitions and constants

Return type *Expression*

eql.**parse_query**(*text*, *implied_any=False*, *implied_base=False*, *preprocessor=None*, *subqueries=True*, *pipes=True*, *cli=False*)
Parse a full EQL query with pipes.

Parameters

- **text** (*str*) – EQL source text to parse
- **implied_any** (*bool*) – Allow for event queries to match on any event type when a type is not specified. If enabled, the query `process_name == "cmd.exe"` becomes `any where process_name == "cmd.exe"`
- **implied_base** (*bool*) – Allow for queries to be built with only pipes. Base query becomes 'any where true'
- **subqueries** (*bool*) – Toggle support for subqueries, which are required by `sequence`, `join`, `descendant of`, `child of` and `event of`
- **pipes** (*bool*) – Toggle support for pipes
- **preprocessor** (*PreProcessor*) – Optional preprocessor to expand definitions and constants

Return type *PipedQuery*

eql.**parse_analytic**(*analytic_info*, *preprocessor=None*, ***kwargs*)
Parse an EQL analytic from a dictionary with metadata.

Parameters

- **analytic_info** (*dict*) – EQL dictionary with metadata and a query to convert to an analytic.
- **preprocessor** (*PreProcessor*) – Optional preprocessor to expand definitions and constants
- **kwargs** – Additional arguments to pass to *parse_query()*

Return type *EqlAnalytic*

eql.**parse_analytics**(*analytics*, *preprocessor=None*, ***kwargs*)
Parse EQL analytics from a list of dictionaries.

Parameters

- **analytics** (*list[dict]*) – EQL dictionary with metadata to convert to an analytic.
- **preprocessor** (*PreProcessor*) – Optional preprocessor to expand definitions and constants
- **kwargs** – Additional arguments to pass to *parse_query()*

Return type list[*EqlAnalytic*]

2.3.2 Python Engine

class eql.**PythonEngine**(*config=None*)
Converter from EQL to Python callbacks.

add_custom_function(*name*, *func*)
Load a python function into the EQL engine.

add_output_hook(*f*)
Register a callback to receive events as they are output from the engine.

add_query(*query*)
Convert an analytic and load into the engine.

add_queries(*queries*)
Add multiple queries to the engine.

add_analytic(*analytic*)
Convert an analytic and load into the engine.

add_analytics(*analytics*)
Add multiple analytics to the engine.

finalize()
Send the engine an EOF signal, so that aggregating pipes can finish.

stream_event(*event*)
Stream a single `Event` through the engine.

stream_events(*events*, *finalize=True*)
Stream `Event` objects through the engine.

2.3.3 Abstract Syntax Tree

EQL syntax tree nodes/schema.

class eql.ast.**BaseNode**
This is the base class for all AST nodes.

render(*precedence=None*, ***kwargs*)
Render the AST in the target language.

class eql.ast.**EqlNode**
The base class for all nodes within the event query language.

class eql.ast.**Walker**
Base class that provides functionality for walking abstract syntax trees of eql.BaseNode.

active_node
Get the active context.

autowalk(*node*, **args*, ***kwargs*)
Automatically walk built-in containers.

classmethod camelized(*node_cls*)
Get the camelized name for the class.

current_event_type
Get the active event type while walking.

get_node_method(*node_cls*, *prefix*)
Get the walk method for a node.

iter_node(*node*)
Iterate through a syntax tree.

parent_node
Get the parent context.

register_func(*node_cls*, *func*, *prefix='_walk_'*)
Register a callback function.

set_context(***kwds*)
Push a node onto the context stack.

walk(*node*, **args*, ***kwargs*)
Walk the syntax tree top-down.

class eql.walkers.**RecursiveWalker**
Walker that will recursively walk and transform a tree.

class eql.walkers.**DepthFirstWalker**
Walk an AST bottom up.

class eql.ast.**Expression**
Base class for expressions.

class eql.ast.**Literal**(*value*)
Static value.

class eql.ast.**TimeRange**(*delta*)
EQL node for an interval of time.

class eql.ast.**Field**(*base*, *path=None*)
Variables and paths in scope of the event.

class eql.ast.**Comparison**(*left*, *comparator*, *right*)
Represents a comparison between two values, as in `<expr> <comparator> <expr>`.

Comparison operators include ==, !=, <, <=, >=, and >.

class eql.ast.**InSet**(*expression*, *container*)
Check if the value of a field within an event matches a list of values.

class eql.ast.**And**(*terms*)
Perform a boolean `and` on a list of expressions.

class eql.ast.**Or**(*terms*)
Perform a boolean `or` on a list of expressions.

class eql.ast.**Not**(*term*)
Negate a boolean expression.

class eql.ast.**FunctionCall**(*name*, *arguments*, *as_method=False*)
A call into a user-defined function by name and a list of arguments.

class eql.ast.**EventQuery**(*event_type*, *query*)
Query over a specific event type with a boolean condition.

class eql.ast.**NamedSubquery**(*query_type*, *query*)
Named of queries perform a subquery with a specific type and returns true if the current event is related.

Query Types: - descendant: Returns true if the pid/unique_pid of the event is a descendant of the subquery process - child: Returns true if the pid/unique_pid of the event is a child of the subquery process - event: Returns true if the pid/unique_pid of the event matches the subquery process

class eql.ast.**NamedParams**(*kv=None*)
An EQL node for key-value named parameters.

class eql.ast.**SubqueryBy**(*query*, *params=None*, *join_values=None*)
Node for holding the *EventQuery* and parameters to join on.

class eql.ast.**Join**(*queries*, *close=None*)
Another boolean query that can join multiple events that share common values.

class eql.ast.**Sequence**(*queries*, *params=None*, *close=None*)
Sequence is very similar to join, but enforces an ordering.

Sequence supports the `until` keyword, which indicates an event that causes it to terminate early.

class eql.ast.**PipeCommand**(*arguments=None*)
Base class for an EQL pipe.

class eql.pipes.**ByPipe**(*arguments=None*)
Pipe that takes a value (field, function, etc.) as a key.

class eql.pipes.**HeadPipe**(*arguments=None*)
Node representing the head pipe, analogous to the unix head command.

class eql.pipes.**TailPipe**(*arguments=None*)
Node representing the tail pipe, analogous to the unix tail command.

class eql.pipes.**SortPipe**(*arguments=None*)
Sorts the pipes by field comparisons.

class eql.pipes.**UniquePipe**(*arguments=None*)
Filters events on a per-field basis, and only outputs the first event seen for a field.

class eql.pipes.**CountPipe**(*arguments=None*)
Counts number of events that match a field, or total number of events if none specified.

class eql.pipes.**FilterPipe**(*arguments=None*)
Takes data coming into an existing pipe and filters it further.

class eql.pipes.**UniqueCountPipe**(*arguments=None*)
Returns unique results but adds a count field.

class eql.ast.**PipedQuery**(*first*, *pipes=None*)
List of all the pipes.

class eql.ast.**EqlAnalytic**(*query*, *metadata=None*)
Analytics are the top-level nodes for matching and returning events.

id
Return the ID from metadata.

name
Return the name from metadata.

2.4 Resources

2.4.1 Blogs

- EQL Threat Hunting
- Ransomware, interrupted: Sodinokibi and the supply chain
- Detecting Adversary Tradecraft with Image Load Event Logging and EQL
- EQL's Highway to Shell
- Getting Started with EQL
- EQL For the Masses
- Introducing EQL

2.4.2 Presentations

- BSides DFW 2019: ATT&CKing Koadic with EQL (slides)
- BlackHat 2019: Fantastic Red-Team Attacks and How to Find Them (slides, blog)
- BSides SATX 2019: The Hunter Games: How to Find the Adversary with EQL (slides)

- Circle City Con 2019: The Hunter Games: How to Find the Adversary with EQL (slides)
- Atomic Friday: Endgame on EQL (slides, notebook)
- MITRE ATT&CKcon: From Technique to Detection

2.4.3 Additional Resources

- Event Query Language (docs, code, twitter)
- EQL Analytics Library (docs, code)

2.5 License

- The Event Query Language has an AGPLv3 License.
- The EQL Analytics Library has an MIT License

CHAPTER 3

License

Check the *license*

Python Module Index

e

Index

O

P

R

S

T

U

W