Платформа Радар

Описание специальных функций корреялции

Версия 3.1.0

ООО Пангео Радар

Описание специальных функций для правил корреляции

Благодаря гибкости языка Python Платформа поддерживает широкий список базовых функций. В том числе:

- операции равенства значений, строкового равенства (независимо от регистра значений);
- операции неравенства значений, строкового неравенства (независимо от регистра значений);
- операции больше, больше или равно, меньше, меньше или равно;
- поиск неполного значения;
- сравнение поля (значения) за временной диапазон, фиксация изменений;
- проверка, что значение в поле начинается с определенного значения;
- проверка наличия или отсутствия значения в поле (поле пустое/не пустое);
- проверка, что значение из поля входит или не входит в указанный список или списки;
- проверка наличия или отсутствия определённого поля в событии;
- использование переменных, для хранения промежуточных значений корреляции;
- сравнение полей (значений) между различными полями события;
- конкатенация (склеивание) значений различных полей внутри одного события;
- использование отрицания к определенному условию или группе условий.

Синтаксис правил корреляции

Пример правила

Пример простого правила, создающего инцидент для каждого события, удовлетворяющего критериям отбора:

```
# Регистрация подписки на события
@log_connection.fetch('#.microsoft.windows.os.authentication.#')
def handle_logline(logline):
    # проверка значений полей событий
    if logline.action == "login_fail" and logline.user == "Admin":
        # Создание инцидента
        alert("admin_login", logline, 8.0, logline.asset_info,
              create_incident=True, assign_to_customer=True)
```

Поддержка UTF-8

Кодировка ASCII и UTF-8 поддерживается в данных объектов **logline**, но не в именах правил и ключей маршрутизации.

Строки UTF-8 должны быть определены как строка **utf** (для этого добавляется «**u**» слева, например, `u"fiёld"`), в случае отсутствия пометки код не выведет предупреждение или ошибку и строки не будут совпадать.

Пример:

```
@log_connection.fetch("ascii-text-only-here")
def handle_logline(line):
    _val = line.get(u"fiёld", None) # field не найдет ни одного элемента
    if _val == u"한글":
        print("Matched")
        alert(..)
    elif _val == "한글": # u отсутствует
        print("Не найдет совпадений!")
    else:
        print("Не совпадает или отсутствует")
```

Ключ маршрутизации (routing key)

Ключ маршрутизации представляет собой последовательность из следующих значений разделенных точкой:

- event.category;
- event.subcategory;
- action;
- outcome (если задано; в противном случае заменяется на “none”);
- reason. (если задано; в противном случае заменяется на “none”);
- “none” (для обратной совместимости с предыдущими версиями продукта);
- “none” (для обратной совместимости с предыдущими версиями продукта);
- “none” (для обратной совместимости с предыдущими версиями продукта);
- logsource.vendor;
- logsource.product;

- logsource.application;
- logsource.subsystem.

Примеры:

```
application.dns_answer.answer.success.dns/answer/noerror.none.none.none.suricata
.suricata.suricata.suricata.dns

authentication.group_member_added.modify.success.none.none.none.none.microsoft.w
indows.os.authentication
```

Объект logline

События, которые правило получает из очереди, приходят в виде объекта **logline**, атрибутами которого являются поля нормализованного события.

Пример:

```
{
  "routing_key":"#.kaspersky-sc.mssql.security_center.#",
  "@timestamp":"2020-02-12T14:35:01.242532+00:00",
  "epoch":1581518101.242532,

  "event":{
    "severity":5.0,
    "category":"antivirus",
    "subcategory":"test.pgr.local",
    "description":"description",
    "worker":{"ip":"172.18.0.27", "host":"cd72b9351328"},
    "logsource":{
      "subsystem":"security_center",
      "product":"kaspersky-sc",
      "vendor":"kaspersky",
      "name":"KES",
      "application":"mssql",
      "host":"172.18.0.27",
      "input":"kaspersky-sc"
    },
    "uuid":"6632bcc38b24cde9bae0ed18d0ec43c8"
  }

  "initiator":{
    "host":{"ip":["10.30.25.123"], "hostname":["test2"], "fqdn":
["test.pgr.local"], "geoip":[]    },
    "user":{"domain":"test.pgr.local","name":"test2\\\\testuser2"}
  },

  "target":{
    "service":{"name":"security/av"},
    "threat":{"name":"malware"},
    "file":{
      "path":
["C:\\\\Users\\\\usefulgarbage\\\\Desktop\\\\av72\\\\eicar_com.zip//eicar.com"],
      "hash":{"sha256":["\u003csha256 HASH object @ 0x7f6dd8462be8\u003e"]},
      "name":["Usefulgarbage test signature"]
```

```
    }
  },

  "action":"detect_malware",
  "outcome":{
    "description":"Результат:     Обнаружено: EICAR-Test-File\\r\\nПользователь:
    pgrdc01\\\\btsymzhitov (Активный пользователь)\\r\\nОбъект:
C:\\\\Users\\\\user\\\\Desktop\\\\av72\\\\eicar_com.zip//eicar.com\\r\\nПричина:
    Экспертный анализ\\r\\nДата выпуска баз:     6/20/2019 1:51:00 AM\\r\\nХеш:
  275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabf651fd0f\\r\\n"
  },

  "observer":{
    "host":{"ip":["10.30.25.1"],"hostname":[],"fqdn":[]    },
    "event":{"type":"GNRL_EV_VIRUS_FOUND","id":"9979744"    }
  },

  "reportchain":{
    "collector":{
      "timestamp":"2020-02-12T17:35:01.000000+03:00",
      "host":{"ip":["10.10.100.2"],"hostname":[],"fqdn":[]}
    },
    "relay":{
      "timestamp":"2020-02-12T17:35:01.000000+03:00",
      "host":{"ip":["10.30.25.1"],"hostname":[],"fqdn":[]}
    }
  },

  "raw":"<...>",
}
```

К полям объекта **logline** можно обращаться:

- как к атрибуту объекта `logline.имя_поля`;
- как к полю словаря `logline["имя_поля"]`.

Допускается обращение ко вложенным объектам.

Пример:

```
data = {
    "@timestamp": "2015-01-01 01:00:00.123456+01:00",
    "@epoch": 1420070400.123456,
    "src": "1.2.3.4",
    "geo_info": {
        "country": "Far Far off",
        "position": {
            "lat": 10.0,
            "long": 20.0,
        }
    }
}

logline = Logline(data)

logline.ts
```

```
# datetime.datetime(2015, 1, 1, 1, 0, 0, 123456, tzinfo=tzoffset(None, 3600))

logline.epoch
# 1420070400.123456

logline["src"]
# "1.2.3.4"

logline.geo_info.country
# "Far Far off"

logline.geo_info.color_of_magic
# None

logline.bogus
# None

logline.bogus.even_worse
# Ошибка AttributeError

logline["geo_info.country.position.lat"]
# 10.0
```

Отладочная печать значений

Для целей отладки доступна функция **print**, позволяющая выводить значения переменных в стандартный поток вывода.

> Большое количество вызовов **print** негативно влияет на производительность правил – рекомендуется использовать только для отладки.

Регистрация подписки на события

Для объявления подписки на события из очереди событий на корреляцию используется метод **fetch**. В качестве параметра передается шаблон для **routing key**. В шаблоне допускается использовать wildcard `#`.

Пример:

```
@log_connection.fetch('#.microsoft.windows.os.authentication.#')
def handle_logline(logline):
    # вывод в отладочную консоль
    print(logline)
    # дальнейшая обработка события
    # ...
```

Одна функция может регистрировать несколько подписок на события.

Пример:

```
@log_connection.fetch('#.microsoft.windows.os.system.#')
@log_connection.fetch('#.microsoft.windows.os.authentication.#')
@log_connection.fetch('#.suricata.suricata.#')
def handle_logline(logline):
    # вывод в отладочную консоль
    print(logline)
    # дальнейшая обработка события
    # ...
```

Регистрация результата или инцидента

При вызове функции **alert** формируется результат работы правила, который записывается в базу данных и отображается в интерфейсе Платформы.

Функция позволяет автоматически создавать инциденты по результатам работы правила и назначать его на ответственную группу пользователей.

Формат вызова функции **alert**:

```
alert(
    template_name,
    logline,
    risklevel,
    asset_info,
    create_incident=False,
    assign_to_customer=False,
    incident_identifier=None
)
```

Где:

- **template_name** (*str*) — имя связанного шаблона.
- **logline** (Logline или AggregatedLogline) — результаты работы функций корреляции и связанные события.
- **risklevel** (float или int) — уровень риска, присваиваемый данному результату (0.0 – 10.0).
- **asset_info** (dict) — словарь значений для поиска актива в базе, может состоять из следующих ключей:
    - **ip (обязательный) –** IP-адрес актива;
    - **hostname –** имя хоста;
    - **mac –** MAC-адрес хоста;
- **create_incident** (bool, по умолчанию False) — автоматическое создание инцидента по данному результату. Инцидент будет создан в статусе «Отладка».
- **assign_to_customer** (bool, по умолчанию False) —автоматическое назначение инцидента группе ответственных. Инцидент будет создан в статусе «Новый».
- **incident_identifier** (str) – уникальный идентификатор в рамках типа инцидента для объединения с нужным происшествием (Например имя учетной записи).

Пример:

```
log_connection.fetch(".suricata.suricata")
def handle_logline(logline):
    if logline.superbad:
        asset_info = {
            'ip': logline.src,
            'hostname': logline.hostname
        }
        alert("template", logline, 9.0, asset_info, create_incident=True)
```

Агрегация событий

Применяется для поиска однотипных событий в рамках временного окна.

Базовая агрегация

Объявление функции агрегации событий:

```
Grouper(
    window_list,
    callback=None,
    over=None,
    aggregation=<logmule.windowed_container.Count object>,
    clock='logline',
    skip_empty=False,
    grouper_id=None,
    overlapping=True
)
```

- **window_list** (*list или str*) — один или несколько размеров окна группировки.
- **callback** (*callable или None*) — функция, вызываемая по завершению каждого временного окна. Если здесь передается None, то должен быть вызван метод **register** для регистрации функции обработки результатов.
- **over** (*None**, *tuple**, *list**, *str или callable*) — поля в объектах **logline**, по которым будет осуществляться группировка.
- **aggregation** (*aggregation factory*) — метод агрегации, например, Count или Sum.
- **clock** (*string*) — может иметь следующие значения:
    - “logline”: временное окно в функции группировки событий (grouper) будет перемещаться по временным меткам из событий;
    - ”heartbeat”: функция группировки событий (grouper) будет использовать внутренние часы коррелятора (один псевдообъект **logline** в секунду по текущему времени UTC).
- **skip_empty** (*bool*) — отмена создания окон, не содержащих данных (пустых), если между последовательными объектами **logline** присутствуют промежутки, которые больше, чем удвоенный временной интервал самого большого окна. Этот флаг не действует, если часы настроены на heartbeat, так как в данном случае между объектами logline нет временного промежутка.
- **grouper_id** (*string*) — переопределение имени функции группировки событий (grouper). Имя, присвоенное в этом параметре, будет использоваться в признаке выборки. Это полезно в случае создания списка функций группировки событий (grouper) и присвоения им имен с использованием *enumerate* или любой другой процедуры.

- **overlapping** (*bool*) — если значение True, то несколько окон перекрываются. Например: [«10s», «1m»] — окно «1m» будет запускаться каждые «10s». Если значение False, окно «1m» будет срабатывать только каждую минуту.

Регистрация обработчика

Функция-обработчик регистрируется с помощью метода **register**:

```
grouper = Grouper(["10m", "1h"], over=("src",))

@grouper.register()
def handle_grouped(grouped_logline):
    print(grouped_logline)
```

Результат работы агрегирующей функции

По окончанию временного окна вызывается зарегистрированная функция обработки результатов, которая в качестве параметра получает объект AggregatedLogline:

```
grouper = Grouper(["10m", "1h"], over=("src",))

@grouper.register()
def handle_grouped(aggregated_logline):
    # aggregated_logline – объект класса AggregatedLogline
    # часто используется параметр с именем grouped
    print(aggregated_logline)
```

Атрибуты объекта AggregatedLogline:

- **@complete_windows** — список завершенных окон.
- **@window_list** — список окон агрегации.
- **@over** — выражение over, из декларации Grouper
- **@key** — ключ, сгенерированный выражением over.
- **@value** — агрегированное значение для окна.
- **@loglines** — собранные объекты logline для каждого объявленного окна.

Пример:

```
# Для декларации
grouper = Grouper(
    ("10s", "5m"),
    over=('initiator.host.hostname', 'initiator.host.ip', 'initiator.host.fqdn'),
    aggregation=Count(segment_by=("target.threat.name", "target.file.path",
"initiator.user.name"),
    store_loglines=True)
)

# Формируемый объект AggregatedLogline
{
  "initiator":{...},
  "@complete_windows":["10s"],
  "@epoch":1581518100.0,
  "@timestamp":"2020-02-12T14:35:00+00:00",
  "@over":[
    "initiator.host.hostname",
```

```
    "initiator.host.ip",
    "initiator.host.fqdn"],
  "@key":[["test2"],
    ["172.30.254.123"],
    ["test.pgr.local"]],
  "@values":{
    "5m":{"(u'malware', (u'C:\\eicar_com.zip//eicar.com',),
u'test2\\testuser2')":1},
    "10s":{"(u'malware', (u'C:\\eicar_com.zip//eicar.com',),
u'test2\\testuser2')":1}
  },
  "@loglines":{
    "5m":[...],
    "10s":[...]
  },
  "@window_list":["10s", "5m"],
  "@routing_key":"#.kaspersky-sc.mssql.security_center.#",
  "@extracted_fields":{}
}
```

Добавление событий в функцию агрегации производится с помощью метода **feed**:

```
grouper = Grouper(['5m'], handle_grouped)

@log_connection.fetch('#.microsoft.windows.os.authentication.#')
def handle_logline(logline):
     grouper.feed(logline)
```

Если в конфигурации функции группировки событий (grouper) параметр **clock** указан как **logline** – производится сдвиг временного окна согласно timestamp, заданному в **logline**.

Сдвиг временного окна без добавления событий в функцию агрегации (функция drive)

Если корреляция строится по событиям, которые случаются редко, рекомендуется использовать сдвиг временного окна по данным из других событий.

Для сдвига временного окна используется функция **drive**:

```
grouper = Grouper(["5m"], handle_grouped)

@log_connection.fetch('#.microsoft.windows.os.authentication.#')
def handle_logline(logline):
    if logline.event_id == "что-то редкое":
        grouper.feed(logline)
    else:
        grouper.drive(logline)
```

Использование системного таймера для сдвига временного окна (heartbeat)

Для корреляции событий, которые приходят неупорядоченно или с некорректными полями @timestamp/epoch необходимо использовать системный таймер.
Использование системного таймера должно быть объявлено в функции группировки событий (grouper) — указать атрибут «heartbeat» в качестве атрибута часов (по умолчанию **logline**).
Это приведет к тому, что агрегирование будет управляться внутренними часами, а не некорректными входящими полями времени.

Внутренний ключ маршрутизации **logmule.heartbeat** используется для получения специального элемента **logline** — **heartbeat** — который генерируется каждую секунду.
При передаче функции группировки событий (grouper) с помощью **heartbeat** вместо обычного **logline** функция группировки событий (grouper) использует временные поля **heartbeat** для управления процессом.

Пример:

```
# clock="heartbeat" - использование системного таймера для сдвига окна
grouper = Grouper("10s", over=("some-key",), clock="heartbeat")

# подписка на события, которые нас интересуют в рамках правила
@log_connection.fetch("#.microsoft.windows.os.authentication.#")
def handle_logline(logline):
    grouper.feed(logline)

# подписка на события системного таймера
@log_connection.fetch("logmule.heartbeat")
def handle_heartbeat(heartbeat):
    # свиг временного окна
    grouper.feed(heartbeat)
```

Для сохранения частоты для каждого правила должен быть только один обработчик heartbeat.
Тем не менее Heartbeat можно использовать более чем в одном вызове.

Пример:

```
@log_connection.fetch("some-route")
def handle_logline_a(logline):
    pass

@log_connection.fetch("different-route")
def handle_logline_b(logline):
    pass

@log_connection.fetch("logmule.heartbeat")
def handle_heartbeat(heartbeat):
    # Оба handle_logline_{a,b} получают heartbeat с периодом 1 секунда
    handle_logline_a(heartbeat)
    handle_logline_b(heartbeat)
```

Примеры использования функции группировки событий (grouper)

Пример:

```
grouper = Grouper(["10m", "1h"], over=("src",))
# Так как Count() является агрегацией по умолчанию, а единственное поле over
# не обязательно записывать как один кортеж, это эквивалентно следующему:
# grouper = Grouper(["10m", "1h"], over="src", aggregation=Count())

@grouper.register()
def handle_grouped(grouped):
    if grouped.value("10m") > 1000 or grouped.value("1h") > 6000:
        risklevel = 6.0
        if grouped.value("10m") > 1000 and grouped.value("1h") > 6000:
            risklevel += 2
        alert("dropped_connections", grouped, risklevel,
              logline.asset_info, create_incident=True)

@log_connection.fetch("firewall")
def handle_logline(logline):
    if (homenet(logline.src) and not homenet(logline.dst) and
        logline.action == "drop"):
        grouper.feed(logline)
```

Обратите внимание, что если поле, переданное в over, содержит списки, то необходимо учитывать порядок элементов. Например, при группировке списка IP-адресов код `['1.1.1.1', '2.2.2.2']` создаст категорию, отличную от категории, заданной с помощью `['2.2.2.2', '1.1.1.1']`. Если такое поведение нежелательно, можно отсортировать список, прежде чем передавать его в функцию агрегации.

При указании нескольких размеров окна, например, `["10m", "1h"]` функция-обработчик будет вызываться каждые 10 минут как для окна «10m», так и для окна «1h». Если указано **overlapping=False**, функция-обработчик будет вызываться каждые 10 минут для окна «10m» и каждый час для окна «1h».

Состояние контекста агрегации может быть сброшено с помощью **grouper.reset ()**. Таким образом контекст будет переведен в исходное состояние до попадания в него объекта **logline**.

Агрегации также могут работать со значениями попадающих в них **logline**. Значения могут быть объединены в результирующий список. Также можно выполнять различные операции над ними: суммировать, вычислять среднее, получать стандартное отклонение или процентили. Для этого необходимо передать в функцию группировки событий (grouper) **logline** либо объект, либо имя функции, возвращающей агрегируемое значение.

Создание списка, содержащего значения из всех объектов **logline** в рамках окна:

```
# Получения списка всех протоколов из обработанных flow
grouper = Grouper("1h", over=("src.ip",),
                  aggregation=Collect("suricata.flow.proto"))

# Суммирование значений поля somefield для всех logline в 1-часовом окне:
# (поле somefield должно быть числовым)
```

```
grouper1 = Grouper("1h", over="src.ip", aggregation=Sum("somefield"))

# Вычисление средней разницы между отправленными и полученными байтами:
def diff(logline):
    return (logline.get("suricata.flow.bytes_sent", 0)
            - logline.get("suricata.flow.bytes_received"))

grouper2 = Grouper("1h", over=("src.ip",), aggregation=Mean(diff))

# Вычисление стандартного отклонения по значениям поля somefield:
grouper3 = Grouper("1h", over=("src.ip",), aggregation=Std("somefield"))

# Получение списка процентилей (1 %, 5 %, 10 %, 50 %, 90 %, 95 %, 99 %) для
собранных значений:
grouper4 = Grouper("1h", over=("src.ip",),
                   aggregation=Percentile("somefield"))
```

Агрегация событий с использованием табличных списков

Все вспомогательные классы и объекты **logline** принимают не только доступ ко вложенным атрибутам, но и вызовы `__getitem__` с использованием точечной нотации. Это позволяет использовать вложенные данные при агрегации, обучении и т.д.

Пример:

```
# Объвления функции агрегации
grouper = Grouper(["10m", "1h"], over=("asset_info.hostname",))

# Регистрация обработчика событий
@grouper.register()
def handle_grouped(grouped):
    pass # do something useful

@log_connection.fetch("firewall")
def handle_logline(logline):
    if (homenet(logline.src) and not homenet(logline.dst) and
        logline.action == "drop"):
        grouper.feed(logline)
```

Агрегирование по ключу

Функция группировки событий (grouper) может разбивать агрегаты по отдельному ключу (предоставляется `segment_by`).

Обратите внимание, что, если поле `segment_by` содержит списки, то порядок элементов имеет значение, поэтому, например, если необходимо разбить список IP-адресов, будет создан сегмент, отличный от созданного.
Если подобный вариант не подходит, можно отсортировать список перед передачей его в функцию группировки событий (grouper). `['1.1.1.1', '2.2.2.2']['2.2.2.2', '1.1.1.1']`

Пример:

```
# Объвления функции агрегации
grouper = Grouper(["10m", "1h"], over=("src.ip",),
                  aggregation=Count(segment_by=("dst.ip",)))

# Регистрация обработчика событий
@grouper.register()
def handle_grouped(grouped):
    grouped.value("10m")
    # == {"8.8.8.8": 20, "8.8.4.4": 7}

@log_connection.fetch()
def handle_logline(logline):
    grouped.feed(logline)
```

Агрегирование по нескольким ключам

Функция группировки событий (grouper) также может одновременно запускать несколько агрегаций в одном потоке **logline**.

Пример:

```
# Объвления функции агрегации
grouper = Grouper(["10m", "1h"], over=("src.ip",),
                  aggregation=Compound(Sum("received_bytes"), Count()))

# Регистрация обработчика событий
@grouper.register()
def handle_grouped(grouped):
    grouped.value("10m")
    # == (243568, 9)

@log_connection.fetch("bro_http")
def handle_logline(logline):
    grouper.feed(logline)
```

Работа с кросс-корреляциями

Контейнер для кросс-корреляций (CorrelationContainer)

Контейнер для кросс-корреляции между источниками. Позволяет обнаруживать цепочки последовательностей атак или последовательности событий, происходящих на одном и том же активе.

Объявление функции корреляции:

```
CorrelationContainer(
    delta,
    distinct_limit,
    callback,
    clock
)
```

Где:

- **delta** (str) — окно корреляции.
- **distinct_limit** (int, по умолчанию "*2*") — требуемое количество различных категорий, которые должны быть отправлены в callback.
- **callback** (*callable*, по умолчанию *None*) — функция, вызываемая по завершению каждого временного окна. Если здесь передается None, то должен быть вызван метод **register** для регистрации функции обработки результатов.
- **clock** (str) — **logline** или **heartbeat**.

Пример:

```
correlation_container = CorrelationContainer("5m")
grouper = Grouper(["1m"], over=("src.ip",))

@correlation_container.register()
def handle_correlated(correlated)
    alert("kill_chain", correlated, 8.0, correlated.asset_info)

@log_connection.fetch("ids_events")
def handle_ids(logline):
    if logline.event.id in (12345, 56789):
        correlation_container.feed("binary download", logline,
                                   key=logline.generate_key("src.ip"))

@grouper.register()
def handle_grouped(grouped)
    if grouped.value("1m") > 200:
        correlation_container.feed("dns burst", grouped, window="1m")
```

Корреляция по цепочке событий (EventChain)

Детектирует заданное количество различных событий для заданного ключа в рамках временного окна.

Объявление функции корреляции:

```
EventChain(
    delta,
    distinct_events=2,
    callback=None
)
```

Где:

- **delta** (str) — окно корреляции.
- **distinct_events** (int) — требуемое количество различных событий.
- **callback** (*callable или None*) — функция, вызываемая по завершению каждого временного окна. Если здесь передается None, то должен быть вызван метод register для регистрации функции обработки результатов.

Функция-обработчик регистрируется с помощью метода **register**:

```
chain = EventChain("12h", distinct_events=2)

@chain.register()
def handle_chained(chained):
    print(chained)
```

В случае успешной корреляции событий вызывается зарегистрированная функция обработки результатов, которая в качестве параметра получает объект AggregatedLogline.

Атрибуты объекта AggregatedLogline:

- **@timestamp** — временная отметка;
- **@epoch** — время в формате epoch;
- **@delta** — заданное временное окно;
- **@events** — перечень значений events;
- **@key** — ключ по которому производилась корреляция;
- **@loglines** — список объектов logline.

Добавление событий в функцию корреляции производится с помощью метода **feed**:

```
feed(event, key, logline, window=None)
```

Где:

- **event** (hashable) — ключ в рамках которого будет считаться разница (например IP-адрес).
- **key** (hashable) — ключ группировки (например имя пользователя).
- **logline** — экземпляр logline.
- **windows** — явное задание окна корреляции.

Пример:

```
# Обнаружение подключений с разных IP-адресов под одной учетной записью

# Объявление функции корреляции
login_chain = EventChain('12h', distinct_events=2)

@log_connection.fetch('#.vpn.#')
def handle_logline(logline):
    if logline.event.subcategory == 'vpn_open':
       login_chain.feed(
           event=','.join(sorted(logline.src_ip)),
           key=logline.target.user.name,
           logline=logline
       )
```

Удаление событий из функции корреляции производится с помощью метода **cancel**:

```
cancel(event, key)
```

Где:

- **event** (hashable) — удаляемое значение. Если задано None будут удалены все значения для key.

- **key** (hashable) — ключ группировки.

Пример:

```
# Обнаружение подключений с разных IP-адресов под одной учетной записью

# Объявление функции корреляции
login_chain = EventChain('12h', distinct_events=2)

@log_connection.fetch('#.vpn.#')
def handle_logline(logline):
    if logline.event.subcategory == 'vpn_open':
       login_chain.feed(
           event=','.join(sorted(logline.src_ip)),
           key=logline.target.user.name,
           logline=logline
       )
    elif logline.event.subcategory == 'vpn_close':
       login_chain.cancel(
           event=','.join(sorted(logline.src_ip)),
           key=logline.target.user.name,
       )
```

Корреляция событий по шаблону (PatternMatcher)

Корреляция событий согласно шаблону появления типов событий в рамках временного окна.

Объявление функции корреляции:

```
PatternMatcher(
    delta,
    pattern,
    callback=None,
    regex=False,
    order_by='@epoch'
)
```

Где:

- **delta** (str) — окно корреляции.
- **pattern** (list или str) —требуемый шаблон для поиска.
- **callback** (*callable или None*) — функция, вызываемая по завершению каждого временного окна. Если здесь передается None, то должен быть вызван метод register для регистрации функции обработки результатов.
- **regex** (bool) — трактовать pattern как регулярное выражение.
- **order_by** (str) — поле по которому выполнять сортировку входящих событий.

Функция-обработчик регистрируется с помощью метода **register**:

```
pattern = PatternMatcher('12h', ['4720'] + ['4726'], order_by="@timestamp")

@pattern.register()
def handle_matched(matched):
    print(matched)
```

Добавление событий в функцию корреляции производится с помощью метода **feed**:

```
feed(event, key, logline, window=None)
```

Где:

- **event** (hashable) — ключ для передачи в шаблон.
- **key** (hashable) — ключ группировки (например имя пользователя).
- **logline** — экземпляр logline.
- **windows** — явное задание окна корреляции.

Пример:

```
# Обнаружение последовательности событий

# Объявление функции корреляции
pattern = PatternMatcher('12h', ['4720'] + ['4726'], order_by="@timestamp")

@log_connection.fetch('#.microsoft.windows.os.system.#')
def handle_logline(logline):
    if logline.get("observer.event.id") in ['4720', '4726']
        key = logline.generate_key('target.user.id')
        pattern.feed(event=event_id, key=key, logline=logline)
```

Удаление событий из функции корреляции производится с помощью метода **cancel**:

```
cancel(event, key)
```

Где:

- **event** (hashable) — удаляемое значение. Если задано None будут удалены все значения для key.
- **key** (hashable) — ключ группировки.

Корреляция по отсутствию события в цепочке (EventTimer)

Используется в случае если требуется обработка ситуации в которой после получения начального события не происходит получение завершающего события в течение заданного временного окна.

Объявление функции корреляции:

```
EventTimer(
    delta,
    start,
    end,
    callback=None,
    interval=30
)
```

Где:

- **delta** (str) — окно корреляции.
- **start** (str) — ключ начального события.
- **end** (str) — ключ начального события.
- **callback** (*callable или None*) — функция, вызываемая по завершению каждого временного окна. Если здесь передается None, то должен быть вызван метод register для регистрации функции обработки результатов.
- **interval** (int) — интервал проверки (секунды).

Функция-обработчик регистрируется с помощью метода **register**:

```
event_timer = EventTimer('5m', start='full scan required', end='resolved')

@event_timer.register()
def handle_results(et_logline):
    print(et_logline)
```

Добавление событий в функцию корреляции производится с помощью метода **feed**:

```
feed(event, key, logline, window=None)
```

Где:

- **event** — ключ для передачи в функцию корреляции.
- **key** — ключ группировки (например имя пользователя).
- **logline** — экземпляр logline.
- **windows** — явное задание окна корреляции.

Пример:

```
# Обнаружения отсутствия завершающего события

# Объявление функции корреляции
event_timer = EventTimer('5m', start='full scan required', end='resolved')

@log_connection.fetch('#.antivirus.antivirus.#')
def handle_logline(logline):

    key = logline.generate_key(
        'initiator.host.hostname',
        'initiator.host.ip',
        'initiator.host.fqdn',
        'initiator.user.name',
        'target.threat.name',
        'target.file.path'
```

```
        )

    event_timer.feed(
        logline.get('reaction.status'),
        key,
        logline
        )
```

Удаление событий из функции корреляции производится с помощью метода **cancel**:

```
cancel(event, key)
```

Где:

- **event** — удаляемое значение. Если задано *None* будут удалены все значения для **key**;
- **key** — ключ группировки.

Руководство по работе с динамическими табличными списками RVS (Rapid Value Store)

Для начала нужно определить коннектор к RVS, для этого используем:

```
my_collection = RVS(collection_name, index[optional])
```

- `collection_name` - str содержащий название коллекции, например "loglines"
- `index` - str содержащий название индекса(обязательного уникального поля), например "id", "ip" [не обязательное поле]

Далее используем созданное подключение для управления коллекцией:

`my_collection.get(pattern[optional])` - получить первый документ в коллекции

- `pattern` - json содержащий условие по которому проводится поиск в формате json, например {"ip":"1.1.1.1"}[не обязательное поле]

получить все документы в коллекции - `my_collection.get_all(pattern[optional])`

- `pattern` - json содержащий условие по которому проводится поиск в формате json, например {"ip":"1.1.1.1"}[не обязательное поле]

поместить документ в коллекцию - `my_collection.set(document[json])`

- `document` - json содержащий документ, который вы хотите добавить в коллекцию в формате json, например {"ip":"1.1.1.1"}

поместить документ в коллекцию с TTL - `my_collection.set_with_ttl(document[json], ttl[int])`

- `document` - json содержащий документ, который вы хотите добавить в коллекцию в формате json, например {"ip":"1.1.1.1"}
- `ttl` - int количество секунд, через которое документ будет удален из коллекции, например 60

добавить индекс в коллекцию - `my_collection.add_index(index[str])`

- `index` - str содержащий название индекса(обязательного уникального поля), например "id", "ip"

показать все открытые коллекции - `my_collection.list()`

отчистить коллекцию от документов(сохранив индексацию) - `my_collection.clear(pattern[optional])`

- `pattern` - json содержащий условие по которому проводится отчитстка (ВСЕ поля подходящие под данный фильтр будут удалены) json, например {"ip":"1.1.1.1"}[не обязательное поле]

удалить один элемент по фильтру (сохранив индексацию) - `my_collection.remove(pattern)`

- `pattern` - json содержащий условие по которому проводится удаление (ПЕРВЫЙ найденный элемент) json, например {"ip":"1.1.1.1"}

удалить коллекцию - `my_collection.drop()`

подсчет количества элементов по фильтру (или всчех элементов в коллекции) - `my_collection.count(pattern[optional])` -

- `pattern` - json содержащий фильтр, например {"ip":"1.1.1.1"}

обновить запись по фильтру - `my_collection.update(pattern)`

- `pattern` - json содержащий фильтр, например {"ip":"1.1.1.1"}

обновить все записи по фильтру - `my_collection.update_all(pattern, data)`

- `pattern` - json содержащий фильтр, например {"ip":"1.1.1.1"}