Платформа Радар

Описание специальных функций корреляции

Версия 3.5.4

ООО Пангео Радар

Оглавление

Оглавление

1. Описание специальных функций для правил корреляции

1.1. Синтаксис правил корреляции

1.1.1. Основные правила

1.1.2. Поддержка UTF-8

1.2. Основной функционал правил корреляции

1.2.1. Ключ маршрутизации `routing key`

1.2.2. Объект `logline`

1.2.3. Отладочная печать значений `print`

1.2.4. Регистрация подписки на события `fetch`

1.2.5. Регистрация результата или инцидента `alert`

1.2.6. Вставка дополнительных полей в результат {#custom_fields}

1.2.7. Настройки правила `rule_settings`

1.2.8. Связанные хранилища значений `stores`

1.2.9. Метод вставки `set`

1.2.10. Функция ограничения отображения количества сырых событий в алерте `trim_result`

1.2.11. Связанные шаблоны `template`

1.2.12. Запись результатов в файл `event_register`

1.3. Агрегация событий

1.3.1. Базовая агрегация `Grouper`

1.3.2. Сдвиг временного окна без добавления событий в функцию агрегации `drive`

1.3.3. Использование системного таймера для сдвига временного окна `heartbeat`

1.3.4. Примеры использования функции группировки событий `grouper`

1.3.5. Агрегация событий с использованием табличных списков

1.3.6. Агрегирование по ключу

1.3.7. Агрегирование по нескольким ключам

1.4. Работа с кросс-корреляциями

1.4.1. Контейнер для кросс-корреляций `CorrelationContainer`

1.4.2. Корреляция по цепочке событий `EventChain`

1.4.3. Корреляция событий по шаблону `PatternMatcher`

1.4.4. Корреляция по отсутствию события в цепочке `EventTimer`

1.5. Использование хранилищ значений и табличных списков

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

1. Описание специальных функций для правил корреляции

Правило корреляции представляет из себя небольшой скрипт на языке Python.

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.

1.1. Синтаксис правил корреляции

1.1.1. Основные правила

Базовый функционал **Python** который необходимо соблюдать:

- Отступы
- Табуляция или пробелы
- Пустые строки
- Использование пробелов
- Соглашения по именованию
- Декораторы
- Метод get()

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)
```

1.1.2. 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")
```

1.2. Основной функционал правил корреляции

1.2.1. 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
```

1.2.2. 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
```

1.2.3. Отладочная печать значений `print`

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.

1.2.4. Регистрация подписки на события `fetch`

Для объявления подписки на события из очереди событий на корреляцию используется метод `fetch`, после чего объявляется функция `handle_logline()`, которая принимает на вход полученное событие.

Пример:

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

В качестве параметра передается шаблон для **routing key**. В шаблоне допускается использовать wildcard `#`.

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
    # ...
```

1.2.5. Регистрация результата или инцидента `alert`

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:

```
@grouper.register()
def handle_grouped(logline):
    asset = build_asset(logline.observer.host.ip,  # IP-адрес актива
                        logline.observer.host.fqdn,  # MAC-адрес хоста
                        logline.observer.host.hostname)  # имя хоста

    alert("template",  # имя связанного шаблона
          logline,  # (Logline или AggregatedLogline - grouped, matched) –
результаты работы функций корреляции и связанные события
          rule_settings["risk_score"],  # уровень риска, присваиваемый данному
результату (0.0 – 10.0)
          asset,  # словарь значений для поиска актива в базе
          create_incident=rule_settings["create_incident"],  # автоматическое
создание инцидента по данному результату
          assign_to_customer=rule_settings["assign_to_customer"],   #
автоматическое назначение инцидента группе ответственных
          incident_identifier=logline.initiator.user.name)  # уникальный
идентификатор в рамках типа инцидента для объединения с нужным происшествием
```

1.2.6. Вставка дополнительных полей в результат {#custom_fields}

В результат, помимо заданных полей, можно вставлять дополнительные поля, перечень которых определен в подразделе "Раздел "Инциденты". Дополнительные поля".
Значение дополнительного поля будет отображаться в карточке инцидента, связанного с результатом обработки правила корреляции.

Для добавления дополнительного поля добавьте строку в правило корреляции `alert`:

```
custom_values={
  "user"="domain\username"
  }
```

Пример:

```
alert("template", logline, 9.0, asset_info, create_incident=True,
    custom_values={
        "user"="domain\username"
    }
)
```

В данном примере "user" - ключ дополнительного поля, "domain\username" - значение дополнительного поля

1.2.7. Настройки правила `rule_settings`

Настройки правила - это глобальная переменная (доступная в любом месте правила), которая объявляется в начале правила с указанием параметров необходимых для работы правила:

```
rule_settings = {
    # основные
    "detection_windows": "5m",  # один или несколько размеров окна группировки
    "risk_score": 8.0,  # оценка риска при создании алерта

    "create_incident": False,  # автоматическое создание инцидента по данному
результату
    "assign_to_customer": False,  # автоматическое назначение инцидента группе
ответственных, инцидент будет создан в статусе «Новый»
    # дополнительные
    "RAW_lines_to_store": 2,  # количество сообщений отображаемых в алерте,
используется совместно с функцией trim_result
    "distinct_events": 2  # требуемое количество различных событий, используется
совместно с функцией EventChain
}
```

1.2.8. Связанные хранилища значений `stores`

Хранилища значений закрепляются за правилом в момент его создания и содержат в себе дополнительную информацию для работы правила:

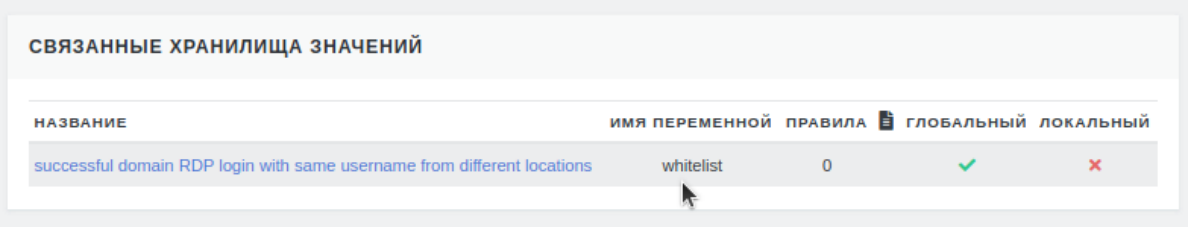

Рисунок 1

```
# username, uuid, location
[
  ['padmin', '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d', 'moscow']
  ['cadmin', 'c1c0a397-add5-475e-9ae6-70cb2e67561b', 'tyumen']
]
```

Доступ к хранилищу производится через объект stores по имени задекларированному при создании правила:

```
whitelist_username = [acc_data[0] for acc_data in stores["whitelist"]]  #
сгенерируется список состоящий из username (['padmin', 'cadmin']), после чего
будет присвоен whitelist_username
whitelist_uuid = [acc_data[1] for acc_data in stores["whitelist"]]  #
сгенерируется список состоящий из uuid (['9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d',
'c1c0a397-add5-475e-9ae6-70cb2e67561b']), после чего будет присвоен
whitelist_uuid
whitelist_location = [acc_data[2] for acc_data in stores["whitelist"]]  #
сгенерируется список состоящий из location (['moscow', 'tyumen']), после чего
будет присвоен  whitelist_location
```

1.2.9. Метод вставки `set`

Метод вставки позволяет добавлять необходимые поля из логлайна в результат работы функции корреляции.

Вызов:

```
@grouper.register ()  # регистрация функции обработки результатов
def handle_grouped(grouped):  # зарегистрированная функция обработки результатов,
которая в качестве параметра получает объект AggregatedLogline (часто
используется параметр с именем grouped)
    all_loglines = grouped.loglines  # список объектов logline

    grouped.set("observer", all_loglines[-1].observer)  # из списка объектов
logline берется последний элемент из которого достается значение observer и
помещается в объект AggregatedLogline по ключу "observer"
    grouped.set("initiator", all_loglines[-1].initiator)  # из списка объектов
logline берется последний элемент из которого достается значение initiator и
помещается в объект AggregatedLogline по ключу "initiator"

    asset = build_asset(...)  # формирование ассета
    alert(...)  # функция регистрации результата или инцидента
```

1.2.10. Функция ограничения отображения количества сырых событий в алерте `trim_result`

Функция ограничения отображения количества сырых событий в алерте.

Объявление и вызов (пример с использованием Grouper - функция агрегации событий):

```
@grouper.register()  # декоратор register()
def handle_grouped(grouped):  # функция обработки результатов
    risk_score = rule_settings["risk_score"]

    # Вызов функции ограничения количества сырых событий в алерте
    trim_result(
        grouped,  # получает объект AggregatedLogline (например grouped или
matched)
        rule_settings["detection_windows"],  # окна группировки
        rule_settings["RAW_lines_to_store"]):  # количество сообщений
отображаемых в алерте
```

1.2.11. Связанные шаблоны `template`

Связанные шаблоны закрепляются за правилом в момент его создания и содержат в себе дополнительную информацию для формирования результата анализа:

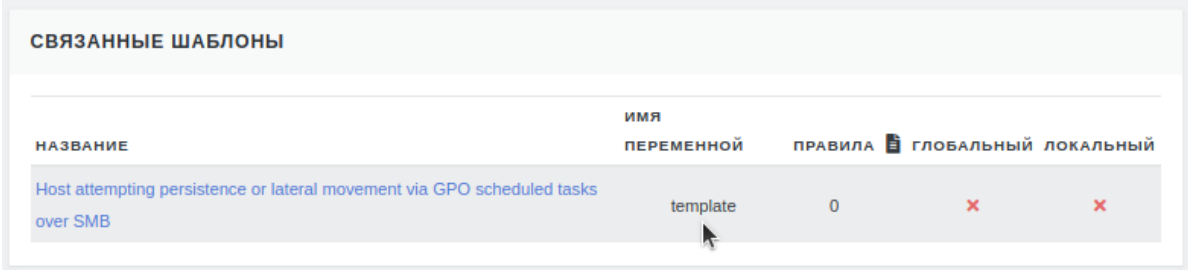

Рисунок 2

```
Пользователь "{{ logline.initiator.user.name }}" (SID: {{
logline.initiator.user.id }}) с узла {{ logline.initiator.host.ip | join(", ") }}
удалённо измененил файл (ScheduledTasks.xml) групповой политики, отвечающего за
управление планировщиком задач Windows.
```

Доступ к связанному шаблону производится через имя шаблона, например `template`.

1.2.12. Запись результатов в файл `event_register`

Функция позволяет выполнить запись результатов работы правил корреляции в файл.

```
event_register({
        "finding_title": "Антивирус – Обнаружено вредоносное ПО",
        "asset_type": "Host",
        "logline_summary": matched.loglines,
        "result_asset_fqdn": matched_logline.target.host.fqdn,
        "result_description": "",
        "result_created_at": matched_logline.reportchain.collector.timestamp,
        "result_id": matched_logline.observer.event.id,
        "result_risk_impact": "",
        "result_incident_identifier": matched_logline.target.threat.name,
        "result_updated_at": matched_logline.reportchain.collector.timestamp,
        "rule_name": "AV-001-Malware detected and not removed Users",
        "result_occurred_at": matched_logline.reportchain.collector.timestamp,
        "correlated_asset_fqdn": matched_logline.target.host.fqdn,
        "result_title": "Обнаружено ВПО в пользовательском сегменте",
        "result_asset_ip": matched_logline.target.host.ip,
        "result_synopsis": "",
        "result_risklevel": rule_settings["risk_score"],
        "result_solution": "",
        "result_analysis_output": "",
        })
```

1.3. Агрегация событий

Применяется для поиска однотипных событий в рамках временного окна.

1.3.1. Базовая агрегация `Grouper`

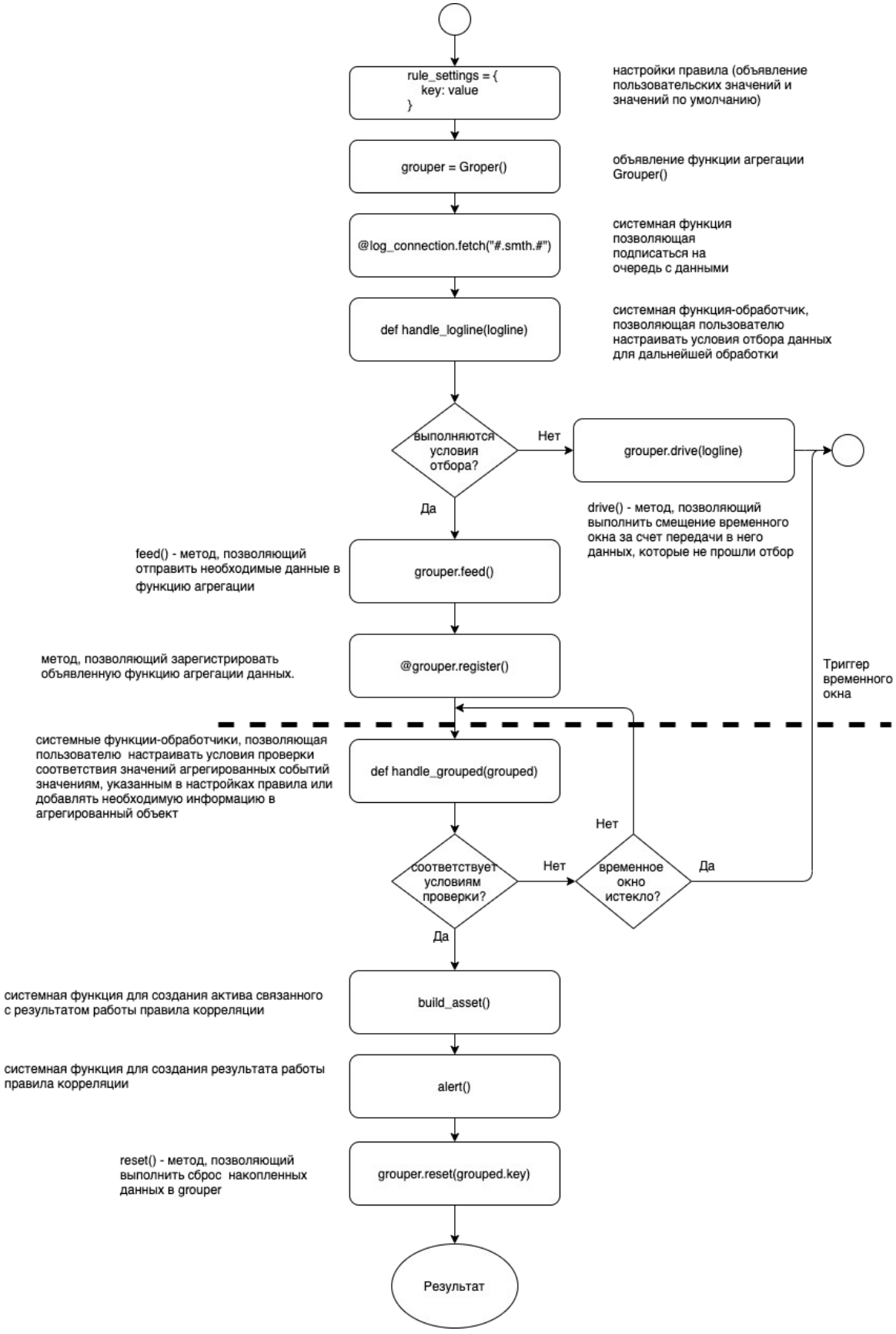

Рисунок 3

Объявление функции агрегации событий:

```
grouper = Grouper(
    rule_settings["detection_windows"],  # размер окна группировки
    over=('initiator.user.name', 'initiator.user.id', 'observer.host.ip',
'observer.host.fqdn', 'observer.host.hostname'),  # поля в объектах logline, по
которым будет осуществляться группировка
    aggregation=Count(store_loglines=True)  # метод агрегации Count считающий
входящие события, параметр store_loglines который колекционирует входящие
события
```

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)
```

Значения из настроек правила

Временное окно - может быть массивом строк из двух значений или строкой из одного значения) Пример задания значений временного окна:

Массив значений, использующийся для скользящего окна внутри большого окна

```
rule_settings = {
    "detection_windows": ("1m","10m")
}
```

Одиночное значение

```
rule_settings = {
    "detection_windows": "10m"
}
```

Могут принимать значения:

- Секунду ("1s")
- Минуты ("1m")
- Часы ("1h")
- Дни ("1d")

over - определение уникальных значений из нормализованных событий, по которым производится подсчет группировка. Может быть одиночное имя поля или массив имен полей.

Пример:

По одному полю:

```
over=("initiator.host.ip")
```

По двум и более:

```
over=("initiator.host.ip", "target.user.name")
```

aggregation - способы агрегации данных.

Агрегация выполняется по полям, перечисленным в параметре over и позволяет выполнять следующие действия над данными:

- Count() - подсчет событий для сгруппированных полей, указанных в поле over. Может использоваться с дополнительные параметрами:
  - segment_by - перечень полей для формирования уникального ключа агрегации
  - store_loglines – необходимо сохранять события или нет. Доступные значения True и False.
- Collect() - получение уникального списка из указанного поля

  Пример:

  `grouper = Grouper("1h", over=("src.ip"), aggregation=Collect("поле, по которому необходимо выполнить расчет "))`
- Sum() - суммирование данных указанного поля

  Пример:

  `grouper = Grouper("1h", over=("src.ip"), aggregation=Sum(поле, по которому необходимо выполнить суммирование))`
- Mean() - расчет среднего

  Пример:

  ```
      return (logline.get("suricata.flow.bytes_sent", 0)
            - logline.get("suricata.flow.bytes_received"))
  grouper = Grouper("1h", over=("src.ip"), aggregation=Mean(diff))```
  ```
- Std() - расчет стандартного отклонения

  Пример:

  `grouper = Grouper("1h", over=("src.ip",), aggregation=Std("поле, по которому необходимо выполнить расчет"))`
- Compound() – параллельный подсчет с применением разных функций подсчета

  Пример:

  ```
                  aggregation=Compound(Sum("received_bytes"), Count()))```
  ```
- clock – параметр отвечает за указание групперу, каким образои осуществлять смещение временного окна. Может принимать следующие значения
  - "logline" -смещение осуществляется с помощью метода drive() за счет объектов logline, которые не прошли отбор для агрегации. Значение по умолчанию.

- "heartbeat" – системный счетчик, который отправляет системное сообщение в группер для принудительного (искусственного) смещения временного окна. Для работы функции необходимо в конце тела правила объявить системную функцию следующего вида:

```
def handle_heartbeat(heartbeat):
    grouper.feed(heartbeat)```
```

Результат работы агрегирующей функции

По окончанию временного окна вызывается зарегистрированная функция обработки результатов, которая в качестве параметра получает объект AggregatedLogline (часто используется параметр с именем `grouped`):

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

Добавление событий в функцию агрегации производится с помощью метода `feed()`:

```
@log_connection.fetch('#.microsoft.windows.os.authentication.#')  # метод для
получения событий на корреляцию
def handle_logline(logline):  # функция принимающая полученные события на
корреляцию
     grouper.feed(logline)
```

Если корреляция строится по событиям, которые случаются редко, рекомендуется использовать сдвиг временного окна по данным из других событий. Для сдвига временного окна используется метод `drive()`:

```
@log_connection.fetch('#.microsoft.windows.os.authentication.#')  # метод для
получения событий на корреляцию
def handle_logline(logline):  # функция принимающая полученные события на
корреляцию
    if logline.event_id == "что-то редкое":
        grouper.feed(logline)
    else:
        grouper.drive(logline)
```

Если при объявлении группера указаны два 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».

```
grouper = Grouper(
    [“10m”, “1h”],  # размеры окон группировки
    over=('initiator.user.name', 'initiator.user.id', 'observer.host.ip',
'observer.host.fqdn', 'observer.host.hostname'),  # поля в объектах logline, по
которым будет осуществляться группировка
    aggregation=Count(store_loglines=True),  # метод агрегации
    overlapping=False)
```

Состояние контекста агрегации может быть сброшено с помощью метода `reset()`:

```
grouper.reset()
```

Таким образом контекст будет переведен в исходное состояние до попадания в него объекта `logline`:

```
@grouper.register()
def handle_grouped(grouped):
    asset = build_asset(...)  # формирование ассета
    alert(...)  # функция регистрации результата или инцидента

    grouper.reset(grouped.key)
```

1.3.2. Сдвиг временного окна без добавления событий в функцию агрегации `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)
```

1.3.3. 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` — `eartbeat` — 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)
```

1.3.4. 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"))
```

1.3.5. 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)
```

1.3.6. 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)
```

1.3.7. 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)
```

1.4. Работа с кросс-корреляциями

1.4.1. 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")
```

1.4.2. Корреляция по цепочке событий `EventChain`

Функция корреляции различных событий детектирует заданное количество различных событий для заданного ключа в рамках временного окна.

Объявление:

```
event_chain = EventChain(
    rule_settings['detection_windows'],  # размер окна группировки
    rule_settings['distinct_events'])  # требуемое количество различных событий
```

Регистрация осуществляется с помощью декоратора `register()`:

```
@event_chain.register()
```

В случае успешной корреляции событий вызывается зарегистрированная функция обработки результатов, которая в качестве параметра получает объект `AggregatedLogline` (часто используется параметр с именем `matched`)

```
@event_chain.register()
def handle_matched_pattern(matched):
    print(matched)
```

Добавление событий в функцию корреляции производится с помощью метода `feed()`:

```
@log_connection.fetch('#.microsoft.windows.os.authentication.#')  # метод для
получения событий на корреляцию
def handle_logline(logline):  # функция принимающая полученные события на
корреляцию

     # Генерируется ключ, по которому определяется разница между событиями
     event_logline = logline.generate_key('initiator.host.ip',
'initiator.host.hostname', 'initiator.host.fqdn')

     # Определяется ключ, который должен быть идентичный в событиях
     event_key = logline.target.user.id

     event_chain.feed(event=event_logline, key=event_key, logline=logline)
```

Удаление событий из функции корреляции производится с помощью метода `cancel()`:

```
@log_connection.fetch('#.microsoft.windows.os.authentication.#')
def handle_logline(logline):
     # Генерируется ключ, по которому определяется разница между событиями
     event_logline = logline.generate_key('initiator.host.ip',
'initiator.host.hostname', 'initiator.host.fqdn')

     # Определяется ключ, который должен быть идентичный в событиях
     event_key = logline.target.user.id

     if logline.event.subcategory == 'vpn_open':
         event_chain.feed(event=event_logline, key=event_key, logline=logline)
     elif logline.event.subcategory == 'vpn_close':
         event_chain.cancel(
             event=event_logline,  #  удаляемое значение. Если задано None будут
удалены все значения для key
             key=event_key)  # ключ группировки
```

1.4.3. Корреляция событий по шаблону `PatternMatcher`

Корреляция событий согласно шаблону появления типов событий в рамках временного окна. Возможно создавать проверку из нескольких `PatternMatcher`.

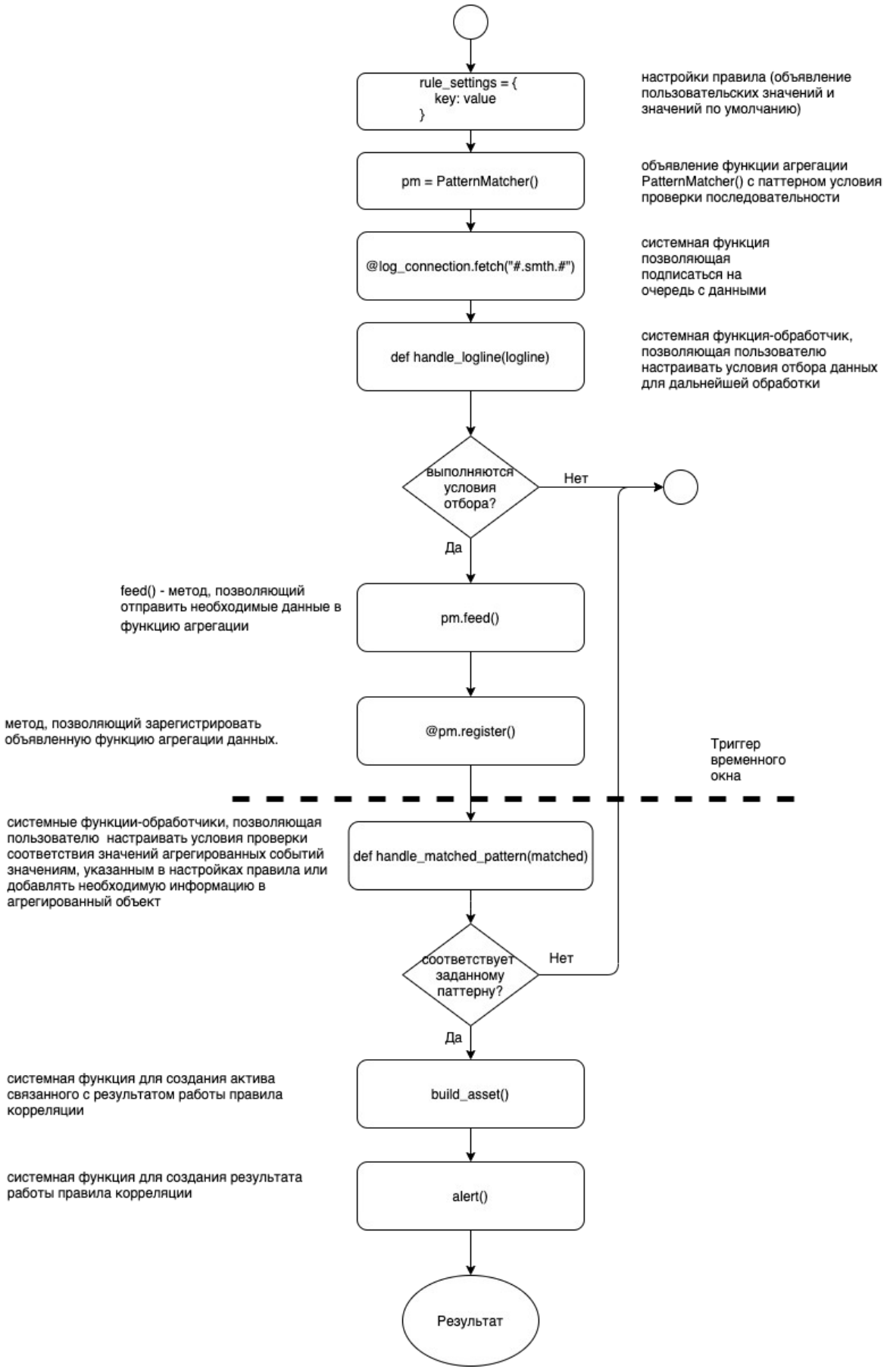

Рисунок 4

Объявление:

```
pattern_matcher = PatternMatcher(
    rule_settings['detection_windows'],  # размер окна группировки
    ['4624'] + ['4672'],  # требуемый шаблон для поиска
    order_by='@timestamp')  # поле по которому выполнять сортировку входящих
событий
```

Доступные параметры:

- order_by - поле из нормализованного сортировка по которому будет осуществляться проверка последовательности.
- Regex – включение директивы использования регулярных выражений в паттернах выявления последовательностей. Доступные значения `True` и `False`.

Регистрация осуществляется с помощью декоратора `register()`:

```
@pattern_matcher.register()
```

В случае успешной корреляции событий вызывается зарегистрированная функция обработки результатов, которая в качестве параметра получает объект `AggregatedLogline` (часто используется параметр с именем `matched`)

```
@pattern_matcher.register()
def handle_matched_pattern(matched):
    print(matched)
```

Добавление событий в функцию корреляции производится с помощью метода `feed()`:

```
@log_connection.fetch('#.microsoft.windows.os.authentication.#')  # метод для
получения событий на корреляцию
def handle_logline(logline):  # функция принимающая полученные события на
корреляцию

    if logline.get("observer.event.id") in ['4624', '4672']:
        # Преобразование составного ключа, который должен совпадать у обоих
событий
        key_user =
logline.generate_key('target.user.id','target.session.id','observer.host.fqdn')

        pattern_matcher.feed(
            event=logline.observer.event.id,  # ключ для передачи в шаблон
            key=key_user,  # ключ группировки (например имя пользователя)
            logline=logline)
```

Удаление событий из функции корреляции производится с помощью метода `cancel()`:

```
pattern_matcher.cancel(
    event=event_logline,  #  удаляемое значение. Если задано None будут удалены
все значения для key
    key=event_key)  # ключ группировки
```

Примеры использования регулярных выражений в паттернах:

Передача паттерна в виде массива

```
PatternMatcher('30m', ['login', 'login'], order_by='custom_field')
```

Передача паттерна в виде сложения массивов

```
PatternMatcher('30m', ['login'] + ['login'], order_by='custom_field')
```

Пример необходимости передачи паттерна через сложение массивов:

```
N = 3
codes = {4531: 'failure', 4221: 'success'}
pattern = ['failure'] * N + ['success']
pm = PatternMatcher('30m', pattern, order_by='custom_field')
if code in codes:
    pm.feed(codes[code], user, logline)
```

Совпадение 'failure', что-то и 'success'

```
pattern = ['failure', '', 'success']
```

Совпадение 'failure', что-то за исключением 'failure' and 'success'

```
pattern = ['failure', '(?!failure)', 'success']
```

Совпадение 'failure', 'failure' или 'other' and 'success'

```
pattern = ['failure', '(failure|other)', 'success']
```

Совпадение 'success' followed by anything except 'success'

```
pattern = ['success', '(?!success)']
```

Пример объявления `PatternMatcher` с использование регулярного выражения:

```
pm = PatternMatcher("1h", pattern, regex=True)
```

1.4.4. Корреляция по отсутствию события в цепочке `EventTimer`

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

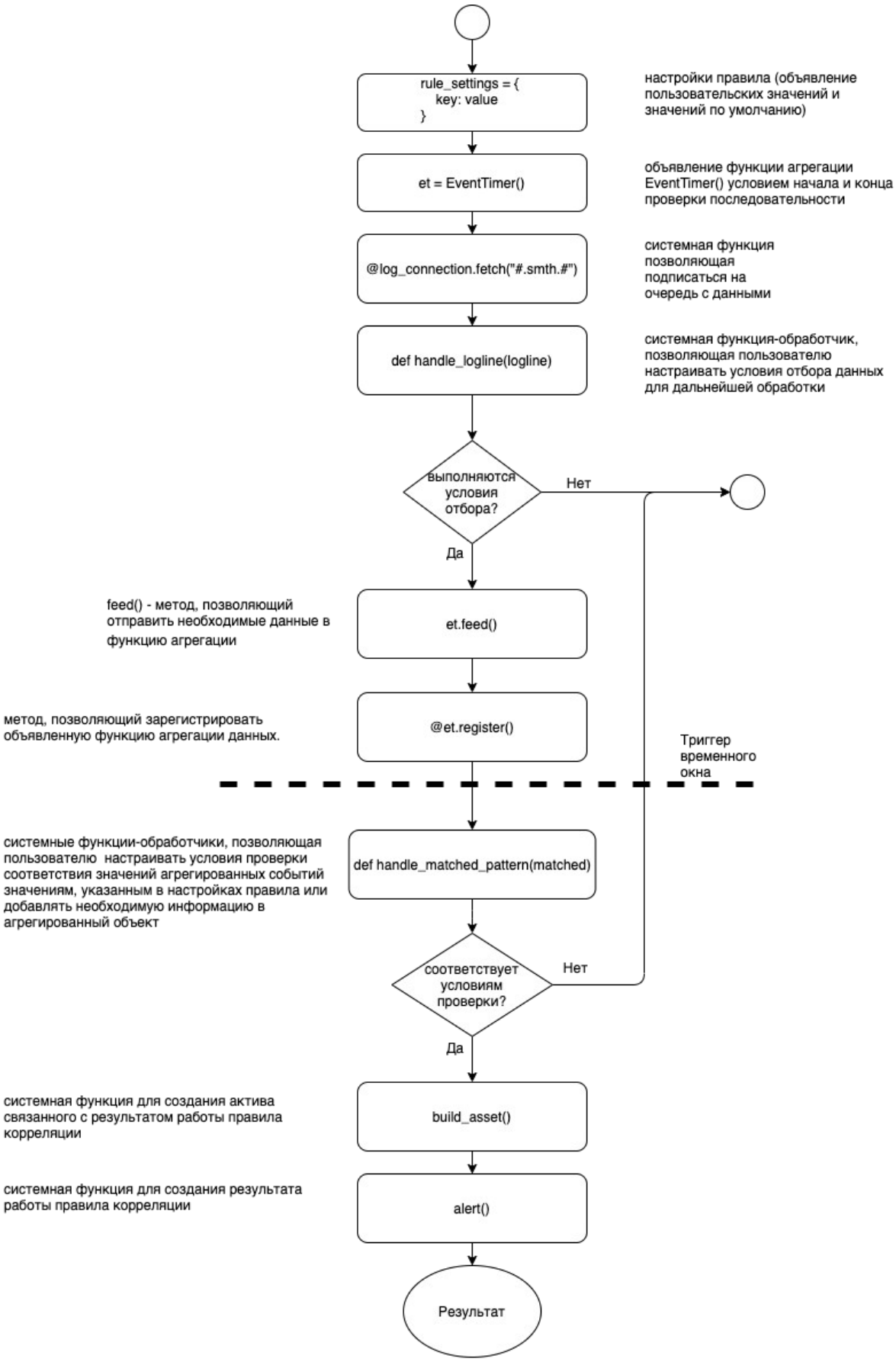

Рисунок 5

Объявление:

```
event_timer = EventTimer(
    rule_settings["detection_windows"],  # размер окна группировки
    start="start",  # ключ начального события
    end="finish",  # ключ конечного события
    interval=30)  # интервал проверки (секунды)
```

Регистрация осуществляется с помощью декоратора `register()`:

```
@event_timer.register()
```

В случае успешной корреляции событий вызывается зарегистрированная функция обработки результатов, которая в качестве параметра получает объект `AggregatedLogline` (часто используется параметр с именем `matched`)

```
@pattern_matcher.register()
def handle_matched_pattern(matched):
    print(matched)
```

Добавление событий в функцию корреляции производится с помощью метода `feed()`:

```
@log_connection.fetch('#.microsoft.windows.os.authentication.#')  # метод для
получения событий на корреляцию
def handle_logline(logline):  # функция принимающая полученные события на
корреляцию
    key = logline.generate_key("ip", "target.threat.name", "path")  # ключ
группировки
    if logline.reaction.executed.name == "detect":
        event_timer.feed("start", key, logline)
    else:
        event_timer.feed("finish", key, logline)
```

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

```
event_timer.cancel(
    event=event_logline,  #  удаляемое значение. Если задано None будут удалены
все значения для key
    key=event_key)  # ключ группировки
```

1.5. Использование хранилищ значений и табличных списков

Важной частью разработки правил корреляции является использование постоянных хранилищ и табличных списков.

Хранилища значений чаще всего используются для профилирования правил корреляции к определенным условиям инфраструктуры. К ним можно отнести хранилища, содержащие информацию о сетях, активах критической инфраструктуры, критичные учетные записи и тому подобное.

Содержимое таких хранилищ можно использовать как белый и\или черный список для определенного типа правил корреляции.

Для их создания необходимо в режиме редактирования правила создать новое "Связанное хранилище значений". В поле "Глобальный набор значений" можно объявить следующие типы данных:

- Функции
- Переменные
- Массивы
- Множества

В зависимости от того, что содержится в хранилище, его вызов может осуществляться через:

- Функцию `exec` (для хранилищ, в которых содержатся функции и переменные)
- Обращение к хранилищу напрямую через функцию `stores` (для хранилищ, в которых содержатся массивы и множества)

Ниже представлены примеры вызова для каждого из описанных вариантов:

Вызов хранилища с описанием сегментов сети

```
exec(stores["customer_networks"])
```

Хранилище customer_networks:

```
home_net = Networks("192.168.0.0/16")
dmz_net = Networks("172.168.100.0/24")
```

Вызов хранилища с белым списком учетных записей

```
whitelist_username = [user_data[0] for user_data in stores["whitelist"]]
```

Хранилище whitelist:

```
# Имя пользователя, адрес источника, fqdn источника, адрес получателя, fqdn
получателя
# target.user.name, initiator.host.ip, initiator.host.fqdn, target.host.ip,
target.host.fqdn

[
    ['jump_user', ['192.168.0.1'], ['ts01.demo.local'], ['172.164.0.1'],
['vbs.demo.local']],
    ['jump_user2', ['192.168.0.2'], ['ts01.demo.local2'], ['172.164.0.2'],
['vbs.demo.local2']],
    ['jump_user3', ['192.168.0.3'], ['ts01.demo.local3'], ['172.164.0.3'],
['vbs.demo.local3']],
]
```

Пример вызова и использования активного хранилища в правиле представлен ниже:
Объявление переменной, содержащей данные из RVS хранилища с информацией об активах
assets_info = RVS('assets_info')

Использование переменной, содержащей информацию из хранилища (последнее условие фильтрации):
if logline.observer.event.id in {'4624', '4625'} and logline.event.auth.method.id == '10' and logline.target.user.id.startswith('S-1-5-21') and assets_info.get({"ip": logline.get('initiator.host.ip'),

"groups": "ДМЗ"}):

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

Табличные списки (Rapid Value Store) являются видом активного хранилища (автоматически изменяемого, в зависимости от условий).

Может использоваться для дополнительной фильтрации при работе с обогащением из таких источников как: Active Directory, Активы и т.д. А также для добавления идентификаторов активов и пользователей в "карантин", для исключения повторных сработок до решения инцидента.

Хранилища могут быть созданы вручную и автоматически посредством обработки событий от источника "Kaspersky-SecurityCenter-db-host-activity" и правила "AV_KES_Hosts with old bases and without workable antivirus". А также посредством исполнения скриптов, получающих информацию из Active Directory и активов Платформы.

Для вызова табличных списков в коде правила необходимо использовать функцию RVS.

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" (необязательное 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"}(необязательное 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"}(необязательное 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"} (необязательное 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"}