|  |  |
| --- | --- |
| Национальный исследовательский ядерный университет  «МИФИ» | |
| Отчет  по лабораторной работе  «Добавление модуля анализа промышленного протокола в ПО Suricata» | |
|  | Выполнили: Голуб Светлана  Герасимов Фёдор  Грубач Арсений  Земский Максим  Группа: Б17-505 |
| Москва 2020г. | |

СОДЕРЖАНИЕ

[1. Описание протокола. 3](#_Toc59219120)

[1.1. Кодировки 3](#_Toc59219121)

[1.2. Взаимодействие между клиентом и сервером. 3](#_Toc59219122)

[1.3. Установка защищённого соединения 5](#_Toc59219123)

[1.4. Обмен сообщениями 7](#_Toc59219124)

[1.5. Сервисы 9](#_Toc59219125)

[2. Описание лабораторного стенда 11](#_Toc59219126)

[2.1. Клиент – сервер. 11](#_Toc59219127)

[2.2. Виртуальные машины. 11](#_Toc59219128)

[3. Модуль для ПО Suricata. 12](#_Toc59219129)

[3.1. Файлы app-layer-opcua.[h, c] 12](#_Toc59219130)

[3.2. Файлы detect-opcua-opcuabuf.h, c] 12](#_Toc59219131)

[3.3. Файл opcua.rules 13](#_Toc59219132)

[4. Запуск ПО Suricata. 14](#_Toc59219133)

[ПРИЛОЖЕНИЕ 16](#_Toc59219134)

[A. OPCUA\_server.py 16](#_Toc59219135)

[B. OPCUA\_client.py 17](#_Toc59219136)

[C. app-layer-opcua.h 18](#_Toc59219137)

[D. app-layer-opcua.c 19](#_Toc59219138)

[E. detect-opcua-opcuabuf.h 32](#_Toc59219139)

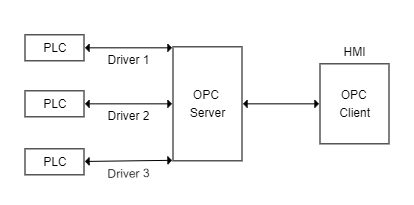
[F. detect-opcua-opcuabuf.c 33](#_Toc59219140)

**OPC UA**

1. **Описание протокола.**

Предшественником OPC UA была спецификация OPC.

**OPC (OLE for Process Control)** — это семейство протоколов и технологий, предоставляющих универсальный механизм сбора данных из различных источников и передачу этих данных любой клиентской программе вне зависимости от типа используемого оборудования.



Однако данная технология была доступна только в операционных системах Microsoft Windows и отсутствовала возможность обеспечить безопасность передачи данных. Это стало причиной разработки Унифицированной архитектуры OPC.

**OPC Unified Architecture (OPC UA)** - спецификация, являющаяся мультиплатформенным стандартом, с реализованной безопасностью. Также система является масштабируемой и поддерживает многопоточность.

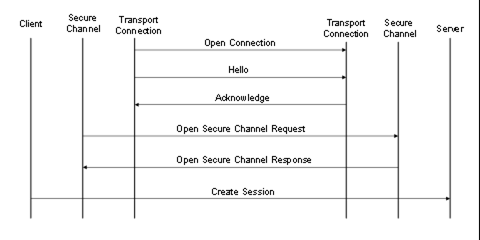
## **Кодировки**

Протокол OPC UA поддерживает передачу данных в трёх кодировках: OPC UA Binary, OPC UA XML и OPC UA JSON. Кодирование и декодирование формата XML занимает много времени, поэтому поддерживается представление информации в виде бинарного файла. Для того, чтобы позволить приложениям OPC UA взаимодействовать с приложениями в сети Интернет, было разработано кодирование данных в формате JSON.

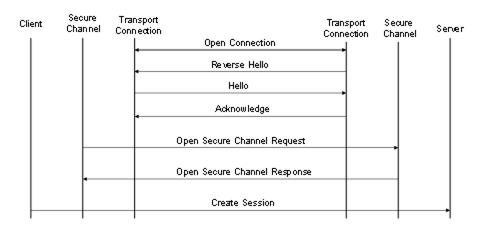
## **Взаимодействие между клиентом и сервером.**

Для начала взаимодействия между клиентом и сервером необходимо установить соединение:

1. Соединение, инициированное клиентом



1. Соединение, инициированное сервером



Форматы сообщений, необходимых для инициации соединения - OPC UA Connection Protocol:

1. **Hello Message** – маркер начала передачи данных от клиента.

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| ProtocolVersion | UInt32 | Последняя версия протокола UACP, поддерживаемая клиентом. |
| ReceiveBufferSize | UInt32 | Самый большой MessageChunk, который может получить отправитель. |
| SendBufferSize | UInt32 | Самый большой MessageChunk, который может отправит отправитель. |
| MaxMessageSize | UInt32 | Максимальный размер любого ответного сообщения. |
| MaxChunkCount | UInt32 | Максимальное количество фрагментов в любом ответном сообщении. |
| EndpointUrl | String | URL-адрес конечной точки, к которой клиент хотел подключиться. |

1. **ReverseHello Message** – маркер начала передачи данных от сервера.

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| ServerUri | String | ApplicationUri сервера, отправившего сообщение.  Закодированное значение должно быть меньше 4096 байтов. |
| EndpointUrl | String | URL-адрес конечной точки, которую Клиент использует при установке SecureChannel.  Это значение должно быть передано обратно серверу в “Hello Message”. |

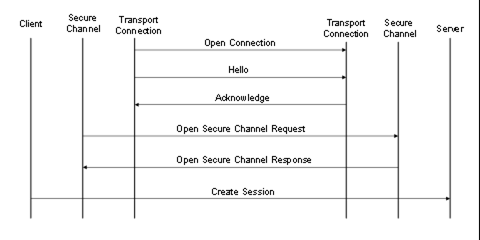
1. **Acknowlege Message** – сообщение, которое сервер посылает клиенту в ответ на сообщение Hello.

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| ProtocolVersion | UInt32 | Последняя версия протокола UACP, поддерживаемая сервером. |
| ReceiveBufferSize | UInt32 | Самый большой MessageChunk, который может получить отправитель.  Это значение не должно быть больше того, что клиент запросил в Hello Message. |
| SendBufferSize | UInt32 | Самый большой MessageChunk, который может отправит отправитель.  Это значение не должно быть больше того, что клиент запросил в Hello Message. |
| MaxMessageSize | UInt32 | Максимальный размер любого сообщения запроса.  Размер сообщения рассчитывается с использованием незашифрованного тела сообщения. |
| MaxChunkCount | UInt32 | Максимальное количество фрагментов в любом сообщении запроса. |

## **Установка защищённого соединения**

После установки соединения клиент посылает сообщение OPEN, которое обозначает открытие канала передачи данных с предложенным клиентом методом шифрования.

В ответ сервер отправляет сообщение OPEN, которое содержит уникальный ID канала передачи данных, а также показывает, что он согласен на предложенный метод шифрования (или его отсутствие).



**OpenSecureChannel Request** - запрос на открытие защищённого канала:

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| requestHeader | RequestHeader | Общие параметры запроса |
| clientCertificate | ApplicationInstanceCertificate | Сертификат, который идентифицирует клиента |
| requestType | Enum SecurityToken RequestType | - ISSUE\_0 создает новый *SecurityToken* для нового *SecureChannel* . - RENEW\_1 создает новый *SecurityToken* для существующего *SecureChannel* |
| secureChannelId | BaseDataType | Идентификатор *SecureChannel,* которому должен принадлежать новый токен. |
| securityMode | Enum MessageSecurityMode | Тип защиты, применяемой к сообщениям. |
| securityPolicyUri | String | URI для *SecurityPolicy* |
| clientNonce | ByteString | Случайное число, которое не должно использоваться ни в каком другом запросе. |
| requestedLifetime | Duration | Запрошенное время жизни в миллисекундах для нового *SecurityToken* |

**OpenSecureChannel Response –** ответ на запрос:

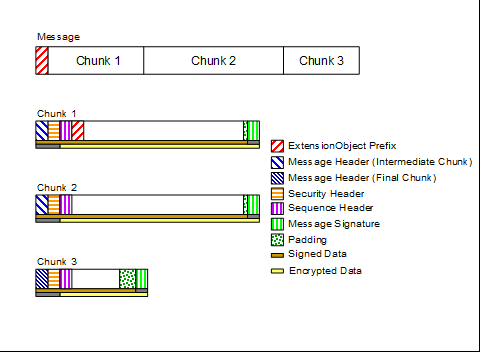
|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| responseHeader | ResponseHeader | Общие параметры ответа |
| securityToken | ChannelSecurityToken | Описывает новый *SecurityToken,* выданный *сервером* |
| channelId | BaseDataType | Уникальный идентификатор *SecureChannel* |
| tokenId | ByteString | Уникальный идентификатор для конкретного *SecurityToken* |
| createdAt | UtcTime | Время создания *SecurityToken* |
| revisedLifetime | Duration | Время жизни *SecurityToken* в миллисекундах |
| serverNonce | ByteString | Случайное число, которое не должно использоваться ни в каком другом запросе |

Протокол поддерживает три режима безопасности: None, SignOnly и SignAndEncrypt.

OPC UA приложения используют сертификаты X.509 v3 для хранения открытых ключей, необходимых для асимметричной криптографии.

## **Обмен сообщениями**

После открытия защищенного канала передачи данных клиент и сервер начинают обмениваться сообщениями MESSAGE (MSG). В силу того, что протоколы транспортного уровня имеют ограниченный размер сегмента, все сообщения разбиваются на фрагменты, размер которых не превышает SendBufferSize, принятый клиентом и сервером при установке соединения.



Каждый фрагмент подписывается и данные, которые идут после заголовка безопасности шифруются выбранным на этапе открытия защищённого канала алгоритмом шифрования (если он предусмотрен).

Фрагменты сообщения имеют следующие заголовки:

1. **Message Header** описывает тип сообщения.

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| MessageType | Byte | Трехбайтовый код ASCII, определяющий тип сообщения:  ·        MSG - Сообщение, защищенное ключами, связанными с каналом.  ·        OPN - OpenSecureChannel.  ·        CLO - CloseSecureChannel. |
| IsFinal | Byte | Указывает, является ли фрагмент последним в сообщении.  ·        C - Промежуточный фрагмент  ·        F - Последний фрагмент  ·        A - Последний фрагмент (используется, когда произошла ошибка и сообщение было прервано).  Это поле имеет значение только для MessageType из «MSG».  Это поле всегда равно «F» для других типов сообщений. |
| MessageSize | UInt32 | Длина фрагмента в байтах |
| SecureChannel | UInt32 | Уникальный идентификатор SecureChannel, присвоенный Сервером.  Если Сервер получает SecureChannel, который он не распознает, он должен вернуть соответствующую ошибку транспортного уровня. |

1. **Security Header**

* **Asymmetric algorithm Security header –** заголовок безопасности в случае асимметричного шифрования.

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| SecurityPolicyUriLength | Int32 | Длина SecurityPolicyUri в байтах.  Это значение не должно превышать 255 байтов.  Если URI не указан, это значение может быть 0 или -1 |
| SecurityPolicyUri | Байт | URI политики безопасности, используемой для защиты сообщения |
| SenderCertificateLength | Int32 | Длина SenderCertificate в байтах  Если сертификат не указан, это значение может быть 0 или -1. |
| SenderCertificate | Байт | X.509 v3 сертификата присваивается отправляющим приложением экземпляра |
| ReceiverCertificateThumbprintLength | Int32 | Длина ReceiverCertificateThumbprint в байтах.  В зашифрованном виде длина этого поля составляет 20 байт. |
| ReceiverCertificateThumbprint | Байт | Индивидуальный код X.509 v3.  Это указывает, какой открытый ключ был использован для шифрования фрагмента.  Это поле должно быть пустым, если сообщение не зашифровано. |

* **Symmetric algorithm Security header –** заголовок безопасности в случае симметричного шифрования.

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| TokenID | UInt32 | Уникальный идентификатор SecureChannel SecurityToken, используемый для защиты сообщения. |

1. **Sequence header –** заголовокгарантирует, что первый зашифрованный фрагмент каждого сообщения, отправляемого по каналу, будет начинаться с разных данных.

|  |  |  |
| --- | --- | --- |
| Имя | Тип данных | Описание |
| SequenceNumber | UInt32 | Монотонно увеличивающийся порядковый номер, присвоенный отправителем каждому фрагменту, отправляемому по защищенному каналу. |
| RequestId | UInt32 | Идентификатор, присвоенный клиентом сообщению запроса OPC UA. Все фрагменты для запроса и связанного с ним ответа используют один и тот же идентификатор. |

## **Сервисы**

В OPC UA всё дальнейшее взаимодействие между клиентом и сервером основано на вызовах сервисов. Вызовы сервисов состоят из запроса и ответа.

**RequestHeader –** заголовок запросов:

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| RequestHeader | structure | Общие параметры запросов для всей сессии |
| authenticationToken | Session AuthenticationToken | Секретный идентификатор сессии |
| timestamp | UtcTime | Время, когда был отправлен запрос |
| requestHandle | IntegerId | Идентификатор запроса (возвращается в ответе) |
| returnDiagnostics | UInt32 | Битовая маска, которая идентифицирует вид необходимой диагностики, если она предусмотрена |
| auditEntryId | String | Идентификатор записи журнала аудита клиента |
| timeoutHint | UInt32 | Тайм-аут, который может использоваться для отмены длительных операций |
| AdditionalHeader | Extensible Parameter AdditionalHeader | Зарезервировано для использования в будущем |

**ResponseHeader** – заголовок ответов:

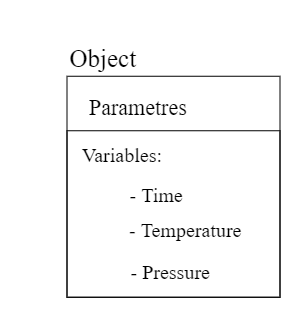
|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| ResponseHeader | structure | Общие параметры ответов для всей сессии |
| timestamp | UtcTime | Время, когда был отправлен ответ |
| requestHandle | IntegerId | Идентификатор запроса, который был получен от клиента |
| serviceResult | StatusCode | Результат вызова службы |
| serviceDiagnostics | DiagnosticInfo | Диагностическая информация, если была запрошена клиентом |
| stringTable[] | String | Параметры диагностической информации |
| additionalHeader | Extensible Parameter AdditionalHeader | Зарезервировано для использования в будущем |

1. **Описание лабораторного стенда**
   1. **Клиент – сервер.**

Для выполнения лабораторной работы была реализована структура клиент-сервер на языке программирования Python 3. Для передачи данных по протоколу OPC UA была использована библиотека freeOPCUA.

В нашем случае в основе адресного пространства OPC UA лежит модель узлов, которая предоставляет серверам удобный способ представления данных клиентам. Объекты и их компоненты представлены в адресном пространстве как набор узлов, описываемых атрибутами и связанных ссылками. Узлы могут быть отдельными объектами, переменными, методами и так дале.

На нашем сервере данные хранятся в качестве переменных объекта Parameters:



Функционал серверного приложения заключается в генерации случайных значений каждые 2 секунды для следующих переменных: температура (в диапазоне от 10 до 50), давление (в диапазоне от 200 до 999) и время (текущее время). Таким образом мы имитируем компоненту АСУ ТП.

Клиент подключается к серверу по порту 4840 и по названию объекта считывает его переменные также каждые 2 секунды.

При получении сигнала прерывания сервер останавливается, прерывая соединение с клиентом. Клиент также прерывает соединение при получении сигнала.

* 1. **Виртуальные машины.**

Роль клиента и сервера выполняют виртуальные машины Kali Linux 2020.3, которые были созданы в программном продукте виртуализации Oracle VM VirtualBox. Две машины были объединены в одну виртуальную сеть следующим образом:

1. В настройках сети VirtualBox была создана сеть Net.
2. В настройках сети виртуальных машин они обе были подключены к сети Net через один из сетевых адаптеров.

На одной из машин был запущен сервер, а на другой клиент. Трафик фиксировался с помощью программного обеспечения WireShark.

1. **Модуль для ПО Suricata.**

Suricata — сетевое средство обнаружения и предотвращения вторжений, имеющее открытый исходный код. Suricata позволяет анализировать множество сетевых протоколов, однако очень малое число промышленных протоколов (например, Modbus). Suricata позволяет добавлять собственноручные модули для расширения списка анализируемых протоколов. В рамках лабораторной работы был реализован модуль для протокола OPC UA.

* 1. **Файлы app-layer-opcua.[h, c]**

Эти файлы отвечают за обнаружение запросов клиента и ответов сервера в трафике.

В них был определён порт для детектирования трафика - 4840 порт (OPC UA).

#define OPCUA\_DEFAULT\_PORT "4840"

На этом уровне есть возможность детектирования пустых сообщений:

{"EMPTY\_MESSAGE", OPCUA\_DECODER\_EVENT\_EMPTY\_MESSAGE}

Для этого в правило необходимо включить следующие ключевые слова:

*app-layer-event:opcua.empty\_message*

* 1. **Файлы detect-opcua-opcuabuf.h, c]**

Эти файлы отвечают за парсинг правил и обнаружение пакетов, соответствующих правилам.

Функции, которые по правилу определяют параметры для проверки пакетов:

- Определение типа сообщения:

static DetectOpcua \***DetectOpcuaTypeParse** (DetectEngineCtx \*de\_ctx, const char \*str)

- Определение размера сообщения:

static DetectOpcua \***DetectOpcuaSizeParse** (DetectEngineCtx \*de\_ctx, const char \*str)

- Определение токена безопасности:

static DetectOpcua \***DetectOpcuaTokenParse** (DetectEngineCtx \*de\_ctx, const char \*str)

- Определяет используемую клиентом функцию:

static DetectOpcua \***DetectOpcuaFunctionParse** (DetectEngineCtx \*de\_ctx,

const char \*str)

- Определяет id запроса:

static DetectOpcua \***DetectOpcuaReqParse** (DetectEngineCtx \*de\_ctx, const char \*str)

Функция, которая проверяет пакеты на соответствие параметрам правила:

static int **DetectOpcuaMatch** (DetectEngineThreadCtx \*det\_ctx, Packet \*p,

const Signature \*s, const SigMatchCtx \*ctx)

В ней происходит парсинг полей поступившего пакета с помощью сдвигов, определённых в начале файла, и сравнение полей со значениями в правилах.

Функция, которая определяет, какое правило будет использовано:

static int DetectOPCUAopcuaSetup(DetectEngineCtx \*de\_ctx, Signature \*s,

const char \*str)

Функция, которая определяет ключевое слово для написания правил:

void DetectOPCUAopcuabufRegister(void)

Она предназначена для того, чтобы зарегистрировать ключевое слово, то есть «объяснить» Suricata, какие функции использовать для парсинга правил.

* 1. **Файл opcua.rules**

В данном файле описаны правила, используемые в работе:

* + - * 1. Детектирование запроса для создания сессии.

alert tcp any any -> any any (msg: "Request to create a session";

opcua: function createSessionReq; sid:1;)

* + - * 1. Детектирование запроса на чтение параметров.

alert tcp any any -> any any (msg: "Request to read parameters";

opcua: function readReq; sid:2;)

* + - * 1. Детектирование открытия нового канала.

alert tcp any any -> any any (msg: "OPN message"; opcua: type OPN;

classtype: bad-unknown; sid:3;)

* + - * 1. Детектирование сообщений HELLO.

alert tcp any any -> any any (msg: "HELLO message"; opcua: type HEL; sid:4;)

* + - * 1. Удаление пакетов, размер которых меньше100.

drop tcp any any -> any any (msg: "Packet size lt 100"; opcua: size lt 100; sid:5;)

* + - * 1. Выявление пакетов, размер которых равен 171.

alert tcp any any -> any any (msg: "Packet size eq 171"; opcua: size eq 171; sid:6;)

* + - * 1. Удаление пакетов, токен безопасности которых недопустим.

drop tcp any any -> any any (msg: "Token has been changed"; opcua: token; sid:7;)

* + - * 1. Выявление запросов, id которых равен 29.

alert tcp any any -> any any (msg: "Request id equal to 29"; opcua: request 29; sid:8;)

* + - * 1. Выявление запроса на чтение параметров.

alert tcp any any -> any any (msg: "Read req"; opcua: function readReq;

classtype: successful-recon-largescale; sid:9;)

* + - * 1. Выявление сообщений типа MSG.

drop tcp any any -> any any (msg: "MSG message"; opcua: type MSG;

classtype: not-suspicious; sid:10;)

Ключевое слово **classtype** определяет приоритет правила.

1. **Запуск ПО Suricata.**
   * 1. Модификация конфигурационного файла **/etc/suricata/suricata.yaml**:
2. Изменение адреса сети, в которой необходимо детектировать трафик.
3. Изменить интерфейс, через который будет проходить трафик.
4. Изменить путь к файлам с правилами для детектирования пакетов.
   * 1. Описание правил для обнаружения пакетов.

* Формат файла - **.rules**
* Он должен находиться в директории, указанной в конфигурационном файле.
* Должен быть прописан сам в конфигурационном файле
  + 1. Запуск ПО Suricata:

sudo suricata -c /etc/suricata/suricata.yaml -i lo --init-errors-fatal

Изображение выглядит как текст

Автоматически созданное описание

- **c** - путь к конфигурационному файлу

- **i** - название интерфейса

-- **init-errors-fatal** - при обнаружении ошибки Suricata завершает работу

* + 1. Запуск сервера.

python3 OPCUA\_server.py

Изображение выглядит как текст

Автоматически созданное описание

* + 1. Запуск клиента.

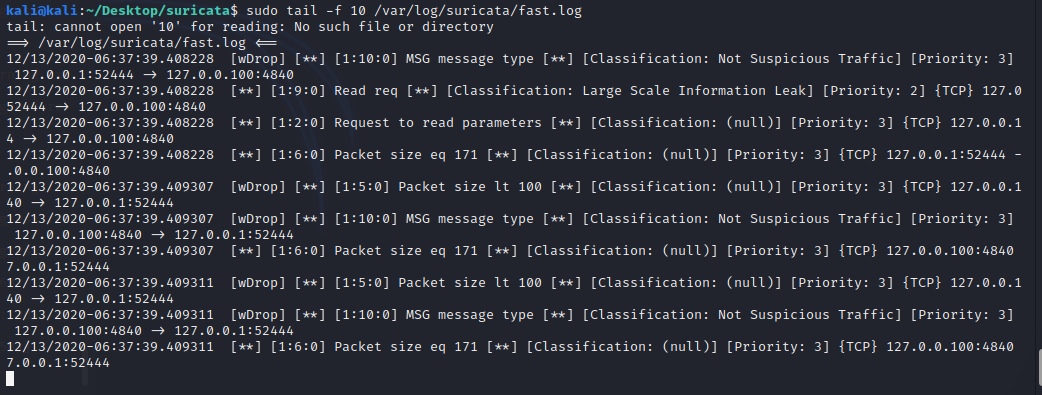
python3 OPCUA\_client.py

Изображение выглядит как текст

Автоматически созданное описание

* + 1. Проверка детектирования пакетов по созданным правилам.

sudo tail -f 10 /var/log/suricata/fast.log



- **f –** просмотрпоследних 10 строк в режиме реального времени.

ПРИЛОЖЕНИЕ

1. **OPCUA\_server.py**

from opcua import Server

from random import randint

import datetime

import time

from opcua import ua

server = Server()

url = "opc.tcp://192.168.56.1:4840"

server.set\_endpoint(url)

name = "OPCUA\_SERVER"

addspace = server.register\_namespace(name)

node = server.get\_objects\_node()

Param = node.add\_object(addspace, "Parameters")

Temp = Param.add\_variable(addspace, "Temperature", 0)

Press = Param.add\_variable(addspace, "Pressure", 0)

Time = Param.add\_variable(addspace, "Time", 0)

Temp.set\_writable()

Press.set\_writable()

Time.set\_writable()

server.start()

print("Server started at {}".format(url))

while True:

try:

Temperature = randint(10, 50)

Pressure = randint(200, 999)

T = datetime.datetime.now()

print(Temperature, Pressure, T)

Temp.set\_value(Temperature)

Press.set\_value(Pressure)

Time.set\_value(T)

time.sleep(2)

except KeyboardInterrupt:

print("Closing connection...")

break

print()

server.stop()

print("Server stopped")

1. **OPCUA\_client.py**

from opcua import Client

import time

import opcua

url = "opc.tcp://192.168.56.1:4840"

client = Client(url)

client.connect()

print("Client connected")

client.load\_type\_definitions()

root = client.get\_root\_node()

print("Root node is: ", root)

objects = client.get\_objects\_node()

print("Objects node is: ", objects)

print("Children of root are: ", root.get\_children())

uri = "OPCUA\_SERVER"

idx = client.get\_namespace\_index(uri)

while True:

try:

Time = root.get\_child(["0:Objects", "{}:Parameters".format(idx), "{}:Time".format(idx)])

Pressure = root.get\_child(["0:Objects", "{}:Parameters".format(idx), "{}:Pressure".format(idx)])

Temp = root.get\_child(["0:Objects", "{}:Parameters".format(idx), "{}:Temperature".format(idx)])

print("Time: ", Time.get\_value())

print("\tPressure: ", Pressure.get\_value())

print("\tTemperature: ", Temp.get\_value())

time.sleep(2)

except KeyboardInterrupt:

print("Closing connection...")

break

client.disconnect()

print("Connection closed")

1. **app-layer-opcua.h**

#ifndef \_\_APP\_LAYER\_OPCUA\_H\_\_

#define \_\_APP\_LAYER\_OPCUA\_H\_\_

#include "detect-engine-state.h"

#include "queue.h"

#include "rust.h"

/\* OPCUA Function Code. \*/

#define OPCUA\_CREATE\_SESSION\_REQ 0xcd

#define OPCUA\_CREATE\_SESSION\_RESP 0xd0

#define OPCUA\_BROWSE\_REQ 0X0f

#define OPCUA\_BROWSE\_RESP 0X12

#define OPCUA\_READ\_REQ 0X77

#define OPCUA\_READ\_RESP 0X7a

void RegisterOPCUAParsers(void);

void OPCUAParserRegisterTests(void);

typedef struct OPCUATransaction

{

/\*\* Internal transaction ID. \*/

uint64\_t tx\_id;

/\*\* Application layer events that occurred

\* while parsing this transaction. \*/

AppLayerDecoderEvents \*decoder\_events;

uint8\_t \*request\_buffer;

uint32\_t request\_buffer\_len;

uint8\_t \*response\_buffer;

uint32\_t response\_buffer\_len;

uint8\_t response\_done; /\*<< Flag to be set when the response is

\* seen. \*/

DetectEngineState \*de\_state;

AppLayerTxData tx\_data;

TAILQ\_ENTRY(OPCUATransaction) next;

} OPCUATransaction;

typedef struct OPCUAState {

/\*\* List of OPCUA transactions associated with this

\* state. \*/

TAILQ\_HEAD(, OPCUATransaction) tx\_list;

/\*\* A count of the number of transactions created. The

\* transaction ID for each transaction is allocted

\* by incrementing this value. \*/

uint64\_t transaction\_max;

} OPCUAState;

#endif /\* \_\_APP\_LAYER\_OPCUA\_H\_\_ \*/

1. **app-layer-opcua.c**

/\*\*

\* \file

\*

\* \author FirstName LastName <yourname@domain>

\*

\* OPCUA application layer detector and parser for learning and

\* opcua pruposes.

\*

\* This opcua implements a simple application layer for something

\* like the echo protocol running on port 7.

\*/

#include "suricata-common.h"

#include "stream.h"

#include "conf.h"

#include "app-layer-detect-proto.h"

#include "app-layer-parser.h"

#include "app-layer-opcua.h"

#include "util-unittest.h"

#include "util-validate.h"

/\* The default port to probe for echo traffic if not provided in the

\* configuration file. \*/

#define OPCUA\_DEFAULT\_PORT "4840"

/\* The minimum size for a message. For some protocols this might

\* be the size of a header. \*/

#define OPCUA\_MIN\_FRAME\_LEN 1

/\* Enum of app-layer events for the protocol. Normally you might

\* have events for errors in parsing data, like unexpected data being

\* received. For opcua we'll make something up, and log an app-layer

\* level alert if an empty message is received.

\*

\* Example rule:

\*

\* alert opcua any any -> any any (msg:"SURICATA OPCUA empty message"; \

\* app-layer-event:opcua.empty\_message; sid:X; rev:Y;)

\*/

enum {

OPCUA\_DECODER\_EVENT\_EMPTY\_MESSAGE,

};

SCEnumCharMap opcua\_decoder\_event\_table[] = {

{"EMPTY\_MESSAGE", OPCUA\_DECODER\_EVENT\_EMPTY\_MESSAGE},

// event table must be NULL-terminated

{ NULL, -1 },

};

static OPCUATransaction \*OPCUATxAlloc(OPCUAState \*state)

{

OPCUATransaction \*tx = SCCalloc(1, sizeof(OPCUATransaction));

if (unlikely(tx == NULL)) {

return NULL;

}

/\* Increment the transaction ID on the state each time one is

\* allocated. \*/

tx->tx\_id = state->transaction\_max++;

TAILQ\_INSERT\_TAIL(&state->tx\_list, tx, next);

return tx;

}

static void OPCUATxFree(void \*txv)

{

OPCUATransaction \*tx = txv;

if (tx->request\_buffer != NULL) {

SCFree(tx->request\_buffer);

}

if (tx->response\_buffer != NULL) {

SCFree(tx->response\_buffer);

}

AppLayerDecoderEventsFreeEvents(&tx->decoder\_events);

SCFree(tx);

}

static void \*OPCUAStateAlloc(void \*orig\_state, AppProto proto\_orig)

{

SCLogNotice("Allocating opcua state.");

OPCUAState \*state = SCCalloc(1, sizeof(OPCUAState));

if (unlikely(state == NULL)) {

return NULL;

}

TAILQ\_INIT(&state->tx\_list);

return state;

}

static void OPCUAStateFree(void \*state)

{

OPCUAState \*opcua\_state = state;

OPCUATransaction \*tx;

SCLogNotice("Freeing opcua state.");

while ((tx = TAILQ\_FIRST(&opcua\_state->tx\_list)) != NULL) {

TAILQ\_REMOVE(&opcua\_state->tx\_list, tx, next);

OPCUATxFree(tx);

}

SCFree(opcua\_state);

}

/\*\*

\* \brief Callback from the application layer to have a transaction freed.

\*

\* \param state a void pointer to the OPCUAState object.

\* \param tx\_id the transaction ID to free.

\*/

static void OPCUAStateTxFree(void \*statev, uint64\_t tx\_id)

{

OPCUAState \*state = statev;

OPCUATransaction \*tx = NULL, \*ttx;

SCLogNotice("Freeing transaction %"PRIu64, tx\_id);

TAILQ\_FOREACH\_SAFE(tx, &state->tx\_list, next, ttx) {

/\* Continue if this is not the transaction we are looking

\* for. \*/

if (tx->tx\_id != tx\_id) {

continue;

}

/\* Remove and free the transaction. \*/

TAILQ\_REMOVE(&state->tx\_list, tx, next);

OPCUATxFree(tx);

return;

}

SCLogNotice("Transaction %"PRIu64" not found.", tx\_id);

}

static int OPCUAStateGetEventInfo(const char \*event\_name, int \*event\_id,

AppLayerEventType \*event\_type)

{

\*event\_id = SCMapEnumNameToValue(event\_name, opcua\_decoder\_event\_table);

if (\*event\_id == -1) {

SCLogError(SC\_ERR\_INVALID\_ENUM\_MAP, "event \"%s\" not present in "

"opcua enum map table.", event\_name);

/\* This should be treated as fatal. \*/

return -1;

}

\*event\_type = APP\_LAYER\_EVENT\_TYPE\_TRANSACTION;

return 0;

}

static int OPCUAStateGetEventInfoById(int event\_id, const char \*\*event\_name,

AppLayerEventType \*event\_type)

{

\*event\_name = SCMapEnumValueToName(event\_id, opcua\_decoder\_event\_table);

if (\*event\_name == NULL) {

SCLogError(SC\_ERR\_INVALID\_ENUM\_MAP, "event \"%d\" not present in "

"opcua enum map table.", event\_id);

/\* This should be treated as fatal. \*/

return -1;

}

\*event\_type = APP\_LAYER\_EVENT\_TYPE\_TRANSACTION;

return 0;

}

static AppLayerDecoderEvents \*OPCUAGetEvents(void \*tx)

{

return ((OPCUATransaction \*)tx)->decoder\_events;

}

/\*\*

\* \brief Probe the input to server to see if it looks like opcua.

\*

\* \retval ALPROTO\_OPCUA if it looks like opcua,

\* ALPROTO\_FAILED, if it is clearly not ALPROTO\_OPCUA,

\* otherwise ALPROTO\_UNKNOWN.

\*/

static AppProto OPCUAProbingParserTs(Flow \*f, uint8\_t direction,

const uint8\_t \*input, uint32\_t input\_len, uint8\_t \*rdir)

{

/\* Very simple test - if there is input, this is opcua. \*/

if (input\_len >= OPCUA\_MIN\_FRAME\_LEN) {

SCLogNotice("Detected as ALPROTO\_OPCUA.");

return ALPROTO\_OPCUA;

}

SCLogNotice("Protocol not detected as ALPROTO\_OPCUA.");

return ALPROTO\_UNKNOWN;

}

/\*\*

\* \brief Probe the input to client to see if it looks like opcua.

\* OPCUAProbingParserTs can be used instead if the protocol

\* is symmetric.

\*

\* \retval ALPROTO\_OPCUA if it looks like opcua,

\* ALPROTO\_FAILED, if it is clearly not ALPROTO\_OPCUA,

\* otherwise ALPROTO\_UNKNOWN.

\*/

static AppProto OPCUAProbingParserTc(Flow \*f, uint8\_t direction,

const uint8\_t \*input, uint32\_t input\_len, uint8\_t \*rdir)

{

/\* Very simple test - if there is input, this is opcua. \*/

if (input\_len >= OPCUA\_MIN\_FRAME\_LEN) {

SCLogNotice("Detected as ALPROTO\_OPCUA.");

return ALPROTO\_OPCUA;

}

SCLogNotice("Protocol not detected as ALPROTO\_OPCUA.");

return ALPROTO\_UNKNOWN;

}

static AppLayerResult OPCUAParseRequest(Flow \*f, void \*statev,

AppLayerParserState \*pstate, const uint8\_t \*input, uint32\_t input\_len,

void \*local\_data, const uint8\_t flags)

{

OPCUAState \*state = statev;

SCLogNotice("Parsing opcua request: len=%"PRIu32, input\_len);

if (input == NULL) {

if (AppLayerParserStateIssetFlag(pstate, APP\_LAYER\_PARSER\_EOF\_TS)) {

/\* This is a signal that the stream is done. Do any

\* cleanup if needed. Usually nothing is required here. \*/

SCReturnStruct(APP\_LAYER\_OK);

} else if (flags & STREAM\_GAP) {

/\* This is a signal that there has been a gap in the

\* stream. This only needs to be handled if gaps were

\* enabled during protocol registration. The input\_len

\* contains the size of the gap. \*/

SCReturnStruct(APP\_LAYER\_OK);

}

/\* This should not happen. If input is NULL, one of the above should be

\* true. \*/

DEBUG\_VALIDATE\_BUG\_ON(true);

SCReturnStruct(APP\_LAYER\_ERROR);

}

/\* Normally you would parse out data here and store it in the

\* transaction object, but as this is echo, we'll just record the

\* request data. \*/

/\* Also, if this protocol may have a "protocol data unit" span

\* multiple chunks of data, which is always a possibility with

\* TCP, you may need to do some buffering here.

\*

\* For the sake of simplicity, buffering is left out here, but

\* even for an echo protocol we may want to buffer until a new

\* line is seen, assuming its text based.

\*/

/\* Allocate a transaction.

\*

\* But note that if a "protocol data unit" is not received in one

\* chunk of data, and the buffering is done on the transaction, we

\* may need to look for the transaction that this newly recieved

\* data belongs to.

\*/

OPCUATransaction \*tx = OPCUATxAlloc(state);

if (unlikely(tx == NULL)) {

SCLogNotice("Failed to allocate new OPCUA tx.");

goto end;

}

SCLogNotice("Allocated OPCUA tx %"PRIu64".", tx->tx\_id);

/\* Make a copy of the request. \*/

tx->request\_buffer = SCCalloc(1, input\_len);

if (unlikely(tx->request\_buffer == NULL)) {

goto end;

}

memcpy(tx->request\_buffer, input, input\_len);

tx->request\_buffer\_len = input\_len;

/\* Here we check for an empty message and create an app-layer

\* event. \*/

if ((input\_len == 1 && tx->request\_buffer[0] == '\n') ||

(input\_len == 2 && tx->request\_buffer[0] == '\r')) {

SCLogNotice("Creating event for empty message.");

AppLayerDecoderEventsSetEventRaw(&tx->decoder\_events,

OPCUA\_DECODER\_EVENT\_EMPTY\_MESSAGE);

}

end:

SCReturnStruct(APP\_LAYER\_OK);

}

static AppLayerResult OPCUAParseResponse(Flow \*f, void \*statev, AppLayerParserState \*pstate,

const uint8\_t \*input, uint32\_t input\_len, void \*local\_data,

const uint8\_t flags)

{

OPCUAState \*state = statev;

OPCUATransaction \*tx = NULL, \*ttx;

SCLogNotice("Parsing OPCUA response.");

/\* Likely connection closed, we can just return here. \*/

if ((input == NULL || input\_len == 0) &&

AppLayerParserStateIssetFlag(pstate, APP\_LAYER\_PARSER\_EOF\_TC)) {

SCReturnStruct(APP\_LAYER\_OK);

}

/\* Probably don't want to create a transaction in this case

\* either. \*/

if (input == NULL || input\_len == 0) {

SCReturnStruct(APP\_LAYER\_OK);

}

/\* Look up the existing transaction for this response. In the case

\* of echo, it will be the most recent transaction on the

\* OPCUAState object. \*/

/\* We should just grab the last transaction, but this is to

\* illustrate how you might traverse the transaction list to find

\* the transaction associated with this response. \*/

TAILQ\_FOREACH(ttx, &state->tx\_list, next) {

tx = ttx;

}

if (tx == NULL) {

SCLogNotice("Failed to find transaction for response on state %p.",

state);

goto end;

}

SCLogNotice("Found transaction %"PRIu64" for response on state %p.",

tx->tx\_id, state);

/\* If the protocol requires multiple chunks of data to complete, you may

\* run into the case where you have existing response data.

\*

\* In this case, we just log that there is existing data and free it. But

\* you might want to realloc the buffer and append the data.

\*/

if (tx->response\_buffer != NULL) {

SCLogNotice("WARNING: Transaction already has response data, "

"existing data will be overwritten.");

SCFree(tx->response\_buffer);

}

/\* Make a copy of the response. \*/

tx->response\_buffer = SCCalloc(1, input\_len);

if (unlikely(tx->response\_buffer == NULL)) {

goto end;

}

memcpy(tx->response\_buffer, input, input\_len);

tx->response\_buffer\_len = input\_len;

/\* Set the response\_done flag for transaction state checking in

\* OPCUAGetStateProgress(). \*/

tx->response\_done = 1;

end:

SCReturnStruct(APP\_LAYER\_OK);

}

static uint64\_t OPCUAGetTxCnt(void \*statev)

{

const OPCUAState \*state = statev;

SCLogNotice("Current tx count is %"PRIu64".", state->transaction\_max);

return state->transaction\_max;

}

static void \*OPCUAGetTx(void \*statev, uint64\_t tx\_id)

{

OPCUAState \*state = statev;

OPCUATransaction \*tx;

SCLogNotice("Requested tx ID %"PRIu64".", tx\_id);

TAILQ\_FOREACH(tx, &state->tx\_list, next) {

if (tx->tx\_id == tx\_id) {

SCLogNotice("Transaction %"PRIu64" found, returning tx object %p.",

tx\_id, tx);

return tx;

}

}

SCLogNotice("Transaction ID %"PRIu64" not found.", tx\_id);

return NULL;

}

/\*\*

\* \brief Called by the application layer.

\*

\* In most cases 1 can be returned here.

\*/

static int OPCUAGetAlstateProgressCompletionStatus(uint8\_t direction) {

return 1;

}

/\*\*

\* \brief Return the state of a transaction in a given direction.

\*

\* In the case of the echo protocol, the existence of a transaction

\* means that the request is done. However, some protocols that may

\* need multiple chunks of data to complete the request may need more

\* than just the existence of a transaction for the request to be

\* considered complete.

\*

\* For the response to be considered done, the response for a request

\* needs to be seen. The response\_done flag is set on response for

\* checking here.

\*/

static int OPCUAGetStateProgress(void \*txv, uint8\_t direction)

{

OPCUATransaction \*tx = txv;

SCLogNotice("Transaction progress requested for tx ID %"PRIu64

", direction=0x%02x", tx->tx\_id, direction);

if (direction & STREAM\_TOCLIENT && tx->response\_done) {

return 1;

}

else if (direction & STREAM\_TOSERVER) {

/\* For the opcua, just the existence of the transaction means the

\* request is done. \*/

return 1;

}

return 0;

}

/\*\*

\* \brief retrieve the tx data used for logging, config, detection

\*/

static AppLayerTxData \*OPCUAGetTxData(void \*vtx)

{

OPCUATransaction \*tx = vtx;

return &tx->tx\_data;

}

/\*\*

\* \brief retrieve the detection engine per tx state

\*/

static DetectEngineState \*OPCUAGetTxDetectState(void \*vtx)

{

OPCUATransaction \*tx = vtx;

return tx->de\_state;

}

/\*\*

\* \brief get the detection engine per tx state

\*/

static int OPCUASetTxDetectState(void \*vtx,

DetectEngineState \*s)

{

OPCUATransaction \*tx = vtx;

tx->de\_state = s;

return 0;

}

void RegisterOPCUAParsers(void)

{

const char \*proto\_name = "opcua";

/\* Check if OPCUA TCP detection is enabled. If it does not exist in

\* the configuration file then it will be enabled by default. \*/

if (AppLayerProtoDetectConfProtoDetectionEnabled("tcp", proto\_name)) {

SCLogNotice("OPCUA TCP protocol detection enabled.");

AppLayerProtoDetectRegisterProtocol(ALPROTO\_OPCUA, proto\_name);

if (RunmodeIsUnittests()) {

SCLogNotice("Unittest mode, registeringd default configuration.");

AppLayerProtoDetectPPRegister(IPPROTO\_TCP, OPCUA\_DEFAULT\_PORT,

ALPROTO\_OPCUA, 0, OPCUA\_MIN\_FRAME\_LEN, STREAM\_TOSERVER,

OPCUAProbingParserTs, OPCUAProbingParserTc);

}

else {

if (!AppLayerProtoDetectPPParseConfPorts("tcp", IPPROTO\_TCP,

proto\_name, ALPROTO\_OPCUA, 0, OPCUA\_MIN\_FRAME\_LEN,

OPCUAProbingParserTs, OPCUAProbingParserTc)) {

SCLogNotice("No opcua app-layer configuration, enabling echo"

" detection TCP detection on port %s.",

OPCUA\_DEFAULT\_PORT);

AppLayerProtoDetectPPRegister(IPPROTO\_TCP,

OPCUA\_DEFAULT\_PORT, ALPROTO\_OPCUA, 0,

OPCUA\_MIN\_FRAME\_LEN, STREAM\_TOSERVER,

OPCUAProbingParserTs, OPCUAProbingParserTc);

}

}

}

else {

SCLogNotice("Protocol detecter and parser disabled for OPCUA.");

return;

}

if (AppLayerParserConfParserEnabled("tcp", proto\_name)) {

SCLogNotice("Registering OPCUA protocol parser.");

/\* Register functions for state allocation and freeing. A

\* state is allocated for every new OPCUA flow. \*/

AppLayerParserRegisterStateFuncs(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAStateAlloc, OPCUAStateFree);

/\* Register request parser for parsing frame from server to client. \*/

AppLayerParserRegisterParser(IPPROTO\_TCP, ALPROTO\_OPCUA,

STREAM\_TOSERVER, OPCUAParseRequest);

/\* Register response parser for parsing frames from server to client. \*/

AppLayerParserRegisterParser(IPPROTO\_TCP, ALPROTO\_OPCUA,

STREAM\_TOCLIENT, OPCUAParseResponse);

/\* Register a function to be called by the application layer

\* when a transaction is to be freed. \*/

AppLayerParserRegisterTxFreeFunc(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAStateTxFree);

/\* Register a function to return the current transaction count. \*/

AppLayerParserRegisterGetTxCnt(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAGetTxCnt);

/\* Transaction handling. \*/

AppLayerParserRegisterGetStateProgressCompletionStatus(ALPROTO\_OPCUA,

OPCUAGetAlstateProgressCompletionStatus);

AppLayerParserRegisterGetStateProgressFunc(IPPROTO\_TCP,

ALPROTO\_OPCUA, OPCUAGetStateProgress);

AppLayerParserRegisterGetTx(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAGetTx);

AppLayerParserRegisterTxDataFunc(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAGetTxData);

/\* What is this being registered for? \*/

AppLayerParserRegisterDetectStateFuncs(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAGetTxDetectState, OPCUASetTxDetectState);

AppLayerParserRegisterGetEventInfo(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAStateGetEventInfo);

AppLayerParserRegisterGetEventInfoById(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAStateGetEventInfoById);

AppLayerParserRegisterGetEventsFunc(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAGetEvents);

/\* Leave this is if you parser can handle gaps, otherwise

\* remove. \*/

AppLayerParserRegisterOptionFlags(IPPROTO\_TCP, ALPROTO\_OPCUA,

APP\_LAYER\_PARSER\_OPT\_ACCEPT\_GAPS);

}

else {

SCLogNotice("OPCUA protocol parsing disabled.");

}

#ifdef UNITTESTS

AppLayerParserRegisterProtocolUnittests(IPPROTO\_TCP, ALPROTO\_OPCUA,

OPCUAParserRegisterTests);

#endif

}

#ifdef UNITTESTS

#endif

void OPCUAParserRegisterTests(void)

{

#ifdef UNITTESTS

#endif

}

1. **detect-opcua-opcuabuf.h**

#ifndef \_\_DETECT\_OPCUA\_OPCUABUF\_H\_\_

#define \_\_DETECT\_OPCUA\_OPCUABUF\_H\_\_

#include "app-layer-opcua.h"

typedef struct DetectOpcua\_ {

uint8\_t type; /\*\* < Opcua msg type to match \*/

uint8\_t function; /\*\* < Opcua function to match \*/

uint8\_t compare; /\*\* < Opcua compare word to match \*/

uint8\_t size; /\*\* < Opcua packet size to match \*/

uint8\_t prev\_token; /\*\* < Opcua previous token to match \*/

uint8\_t req; /\*\* < Opcua request id to match \*/

} DetectOpcua;

void DetectOPCUAopcuabufRegister(void);

#endif /\* \_\_DETECT\_OPCUA\_OPCUABUF\_H\_\_ \*/

1. **detect-opcua-opcuabuf.c**

#include "suricata-common.h"

#include "conf.h"

#include "detect.h"

#include "detect-parse.h"

#include "detect-engine.h"

#include "detect-engine-mpm.h"

#include "detect-engine-prefilter.h"

#include "app-layer-opcua.h"

#include "detect-opcua-opcuabuf.h"

#include "util-byte.h"

/\* Offsets \*/

#define OPCUA\_MIN\_FRAME\_LEN 24

#define OPCUA\_SIZE\_OFFSET 4

#define OPCUA\_TOKEN\_OFFSET 12

#define OPCUA\_REQ\_OFFSET 20

#define OPCUA\_FUNC\_OFFSET 26

#define OPCUA\_START\_TOKEN 0X01

/\* For size \*/

#define LESS 0X01

#define EQUAL 0X02

#define GREATER 0X03

/\* OPCUA Function Code. \*/

#define OPCUA\_CREATE\_SESSION\_REQ 0xcd

#define OPCUA\_CREATE\_SESSION\_RESP 0xd0

#define OPCUA\_BROWSE\_REQ 0X0f

#define OPCUA\_BROWSE\_RESP 0X12

#define OPCUA\_READ\_REQ 0X77

#define OPCUA\_READ\_RESP 0X7a

/\* OPCUA msg types \*/

#define OPCUA\_MSG\_TYPE 0x4d

#define OPCUA\_HELLO\_TYPE 0x48

#define OPCUA\_ACK\_TYPE 0x41

#define OPCUA\_OPN\_TYPE 0x4f

/\*\*

\* \brief Regex for parsing the opcua msg type string

\*/

#define PARSE\_REGEX\_TYPE "^\\s\*\"?\\s\*type\\s\*([A-z]+)\\s\*\"?\\s\*$"

static DetectParseRegex type\_parse\_regex;

/\*\*

\* \brief Regex for parsing the opcua msg size string

\*/

#define PARSE\_REGEX\_SIZE "^\\s\*\"?\\s\*size\\s\*(lt|eq|gt)(\\s+(\\d+))\\s\*\"?\\s\*$"

static DetectParseRegex size\_parse\_regex;

/\*\*

\* \brief Regex for parsing the opcua token string

\*/

#define PARSE\_REGEX\_TOKEN "^\\s\*\"?\\s\*token\\s\*\"?\\s\*$"

static DetectParseRegex token\_parse\_regex;

/\*\*

\* \brief Regex for parsing the opcua requestid string

\*/

#define PARSE\_REGEX\_REQ "^\\s\*\"?\\s\*request\\s\*(\\d+)\\s\*\"?\\s\*$"

static DetectParseRegex req\_parse\_regex;

/\*\*

\* \brief Regex for parsing the opcua function string

\*/

#define PARSE\_REGEX\_FUNCTION "^\\s\*\"?\\s\*function\\s\*([A-z]+)\\s\*\"?\\s\*$"

static DetectParseRegex function\_parse\_regex;

static int g\_opcua\_opcuabuf\_id = 0;

static int DetectOPCUAopcuaSetup(DetectEngineCtx \*, Signature \*, const char \*);

#ifdef UNITTESTS

static void DetectOPCUAopcuabufRegisterTests(void);

#endif

/\*\* \internal

\*

\* \brief this function will free memory associated with DetectOpcua

\*

\* \param ptr pointer to DetectOpcua

\*/

static void DetectOpcuaFree(DetectEngineCtx \*de\_ctx, void \*ptr) {

SCEnter();

DetectOpcua \*opcua = (DetectOpcua \*) ptr;

if(opcua) {

SCFree(opcua);

}

}

/\*\* \internal

\*

\* \brief This function is used to parse OPCUA parameters in function mode

\*

\* \param de\_ctx Pointer to the detection engine context

\* \param str Pointer to the user provided option

\*

\* \retval Pointer to DetectOpcuaData on success or NULL on failure

\*/

static DetectOpcua \*DetectOpcuaFunctionParse(DetectEngineCtx \*de\_ctx, const char \*str)

{

SCEnter();

DetectOpcua \*opcua = NULL;

char arg[MAX\_SUBSTRINGS], \*ptr = arg;

int ov[MAX\_SUBSTRINGS], res, ret;

ret = DetectParsePcreExec(&function\_parse\_regex, str, 0, 0, ov, MAX\_SUBSTRINGS);

SCLogNotice("PcreExec function: %d", ret);

if (ret < 1)

goto error;

res = pcre\_copy\_substring(str, ov, MAX\_SUBSTRINGS, 1, arg, MAX\_SUBSTRINGS);

if (res < 0) {

SCLogError(SC\_ERR\_PCRE\_GET\_SUBSTRING, "pcre\_get\_substring failed");

goto error;

}

/\* We have a correct Opcua function option \*/

opcua = (DetectOpcua \*) SCCalloc(1, sizeof(DetectOpcua));

if (unlikely(opcua == NULL))

goto error;

if (strcmp("createSessionReq", ptr) == 0){

opcua->function = OPCUA\_CREATE\_SESSION\_REQ;

opcua->type = OPCUA\_MSG\_TYPE;

}

else if (strcmp("createSessionResp", ptr) == 0){

opcua->function = OPCUA\_CREATE\_SESSION\_RESP;

opcua->type = OPCUA\_MSG\_TYPE;

}

else if (strcmp("browseReq", ptr) == 0){

opcua->function = OPCUA\_BROWSE\_REQ;

opcua->type = OPCUA\_MSG\_TYPE;

}

else if (strcmp("browseResp", ptr) == 0){

opcua->function = OPCUA\_BROWSE\_RESP;

opcua->type = OPCUA\_MSG\_TYPE;

}

else if (strcmp("readReq", ptr) == 0){

opcua->function = OPCUA\_READ\_REQ;

opcua->type = OPCUA\_MSG\_TYPE;

}

else if (strcmp("readResp", ptr) == 0){

opcua->function = OPCUA\_READ\_RESP;

opcua->type = OPCUA\_MSG\_TYPE;

}

else

SCLogError(SC\_ERR\_INVALID\_VALUE, "Invalid value for opcua function: %s", ptr);

SCLogDebug("will look for opcua function %d", opcua->function);

SCReturnPtr(opcua, "DetectOpcuaFunction");

error:

if (opcua != NULL)

DetectOpcuaFree(de\_ctx, opcua);

SCReturnPtr(NULL, "DetectOpcua");

}

/\*\* \internal

\*

\* \brief This function is used to parse OPCUA parameters in type mode

\*

\* \param de\_ctx Pointer to the detection engine context

\* \param str Pointer to the user provided option

\*

\* \retval Pointer to DetectOpcuaData on success or NULL on failure

\*/

static DetectOpcua \*DetectOpcuaTypeParse(DetectEngineCtx \*de\_ctx, const char \*str)

{

SCEnter();

DetectOpcua \*opcua = NULL;

char arg[MAX\_SUBSTRINGS], \*ptr = arg;

int ov[MAX\_SUBSTRINGS], res, ret;

SCLogNotice("Type? %s", str);

ret = DetectParsePcreExec(&type\_parse\_regex, str, 0, 0, ov, MAX\_SUBSTRINGS);

SCLogNotice("PcreExec type: %d", ret);

if (ret < 1)

goto error;

res = pcre\_copy\_substring(str, ov, MAX\_SUBSTRINGS, 1, arg, MAX\_SUBSTRINGS);

if (res < 0) {

SCLogError(SC\_ERR\_PCRE\_GET\_SUBSTRING, "pcre\_get\_substring failed");

goto error;

}

/\* We have a correct Opcua type option \*/

opcua = (DetectOpcua \*) SCCalloc(1, sizeof(DetectOpcua));

if (unlikely(opcua == NULL)){

SCLogNotice("Opcua - NULL");

goto error;

}

if (strcmp("HEL", ptr) == 0)

opcua->type = OPCUA\_HELLO\_TYPE;

else if (strcmp("OPN", ptr) == 0)

opcua->type = OPCUA\_OPN\_TYPE;

else if (strcmp("ACK", ptr) == 0)

opcua->type = OPCUA\_ACK\_TYPE;

else if (strcmp("MSG", ptr) == 0)

opcua->type = OPCUA\_MSG\_TYPE;

else {

SCLogError(SC\_ERR\_INVALID\_VALUE, "Invalid value for opcua msg type: %s", ptr);

goto error;

}

SCLogDebug("will look for opcua type %d", opcua->function);

SCReturnPtr(opcua, "DetectOpcuaType");

error:

if (opcua != NULL)

DetectOpcuaFree(de\_ctx, opcua);

SCReturnPtr(NULL, "DetectOpcua");

}

/\*\* \internal

\*

\* \brief This function is used to parse OPCUA parameters in size mode

\*

\* \param de\_ctx Pointer to the detection engine context

\* \param str Pointer to the user provided option

\*

\* \retval Pointer to DetectOpcuaData on success or NULL on failure

\*/

static DetectOpcua \*DetectOpcuaSizeParse(DetectEngineCtx \*de\_ctx, const char \*str)

{

SCEnter();

DetectOpcua \*opcua = NULL;

char arg[MAX\_SUBSTRINGS], \*ptr = arg;

int ov[MAX\_SUBSTRINGS], res, ret;

SCLogNotice("Size? %s", str);

ret = DetectParsePcreExec(&size\_parse\_regex, str, 0, 0, ov, MAX\_SUBSTRINGS);

SCLogNotice("PcreExec size: %d", ret);

if (ret < 1)

goto error;

res = pcre\_copy\_substring(str, ov, MAX\_SUBSTRINGS, 1, arg, MAX\_SUBSTRINGS);

SCLogNotice("Compare: %s", arg);

if (res < 0) {

SCLogError(SC\_ERR\_PCRE\_GET\_SUBSTRING, "pcre\_get\_substring failed");

goto error;

}

/\* We have a correct Opcua option \*/

opcua = (DetectOpcua \*) SCCalloc(1, sizeof(DetectOpcua));

if (unlikely(opcua == NULL)){

SCLogNotice("Opcua - NULL");

goto error;

}

if (strcmp("lt", ptr) == 0)

opcua->compare = LESS;

else if (strcmp("eq", ptr) == 0)

opcua->compare = EQUAL;

else if (strcmp("gt", ptr) == 0)

opcua->compare = GREATER;

else {

SCLogError(SC\_ERR\_INVALID\_VALUE, "Invalid value: %s", ptr);

goto error;

}

res = pcre\_copy\_substring(str, ov, MAX\_SUBSTRINGS, 3, arg, MAX\_SUBSTRINGS);

SCLogNotice("Size: %s", arg);

if (res < 0) {

SCLogError(SC\_ERR\_PCRE\_GET\_SUBSTRING, "pcre\_get\_substring failed");

goto error;

}

/\* We have a correct Opcua option \*/

opcua = (DetectOpcua \*) SCCalloc(1, sizeof(DetectOpcua));

if (unlikely(opcua == NULL)){

SCLogNotice("Opcua - NULL");

goto error;

}

if (!isdigit((unsigned char)ptr[0]))

goto error;

if (StringParseUint8(&opcua->size, 10, 0, (const char \*)ptr) < 0) {

SCLogNotice("Size: %d", opcua->size);

SCLogError(SC\_ERR\_INVALID\_VALUE, "Invalid value for opcua size: %s", (const char \*)ptr);

goto error;

}

SCLogDebug("will look for opcua size %d", opcua->size);

SCReturnPtr(opcua, "DetectOpcuaType");

error:

if (opcua != NULL)

DetectOpcuaFree(de\_ctx, opcua);

SCReturnPtr(NULL, "DetectOpcua");

}

/\*\* \internal

\*

\* \brief This function is used to parse OPCUA parameters in token mode

\*

\* \param de\_ctx Pointer to the detection engine context

\* \param str Pointer to the user provided option

\*

\* \retval Pointer to DetectOpcuaData on success or NULL on failure

\*/

static DetectOpcua \*DetectOpcuaTokenParse(DetectEngineCtx \*de\_ctx, const char \*str)

{

SCEnter();

DetectOpcua \*opcua = NULL;

int ov[MAX\_SUBSTRINGS], ret;

SCLogNotice("Token? %s", str);

ret = DetectParsePcreExec(&token\_parse\_regex, str, 0, 0, ov, MAX\_SUBSTRINGS);

SCLogNotice("PcreExec token: %d", ret);

if (ret < 1)

goto error;

/\* We have a correct Opcua type option \*/

opcua = (DetectOpcua \*) SCCalloc(1, sizeof(DetectOpcua));

if (unlikely(opcua == NULL)){

SCLogNotice("token Opcua = NULL");

goto error;

}

opcua->prev\_token = OPCUA\_START\_TOKEN;

SCLogDebug("will look for changing opcua token %d", opcua->prev\_token);

SCReturnPtr(opcua, "DetectOpcuaType");

error:

if (opcua != NULL)

DetectOpcuaFree(de\_ctx, opcua);

SCReturnPtr(NULL, "DetectOpcua");

}

/\*\* \internal

\*

\* \brief This function is used to parse OPCUA parameters in request mode

\*

\* \param de\_ctx Pointer to the detection engine context

\* \param str Pointer to the user provided option

\*

\* \retval Pointer to DetectOpcuaData on success or NULL on failure

\*/

static DetectOpcua \*DetectOpcuaReqParse(DetectEngineCtx \*de\_ctx, const char \*str)

{

SCEnter();

DetectOpcua \*opcua = NULL;

char arg[MAX\_SUBSTRINGS], \*ptr = arg;

int ov[MAX\_SUBSTRINGS], res, ret;

SCLogNotice("Request? %s", str);

ret = DetectParsePcreExec(&req\_parse\_regex, str, 0, 0, ov, MAX\_SUBSTRINGS);

SCLogNotice("PcreExec req: %d", ret);

if (ret < 1)

goto error;

res = pcre\_copy\_substring(str, ov, MAX\_SUBSTRINGS, 1, arg, MAX\_SUBSTRINGS);

SCLogNotice("Request id: %s", arg);

if (res < 0) {

SCLogError(SC\_ERR\_PCRE\_GET\_SUBSTRING, "pcre\_get\_substring failed");

goto error;

}

/\* We have a correct Opcua option \*/

opcua = (DetectOpcua \*) SCCalloc(1, sizeof(DetectOpcua));

if (unlikely(opcua == NULL)){

SCLogNotice("Opcua - NULL");

goto error;

}

if (!isdigit((unsigned char)ptr[0]))

goto error;

if (StringParseUint8(&opcua->req, 10, 0, (const char \*)ptr) < 0) {

SCLogNotice("Request id: %d", opcua->req);

SCLogError(SC\_ERR\_INVALID\_VALUE, "Invalid value for opcua request id: %s", (const char \*)ptr);

goto error;

}

SCLogDebug("will look for opcua request id %d", opcua->req);

SCReturnPtr(opcua, "DetectOpcuaType");

error:

if (opcua != NULL)

DetectOpcuaFree(de\_ctx, opcua);

SCReturnPtr(NULL, "DetectOpcua");

}

/\*\*

\* \brief Checks if the packet sent as the argument, has a valid or invalid

\* values.

\*

\* \param det\_ctx Pointer to the detection engine thread context

\* \param p Pointer to the Packet currently being matched

\* \param s Pointer to the Signature, the packet is being currently

\* matched with

\* \param ctx Pointer to the keyword\_structure(SigMatch) from the above

\* Signature, the Packet is being currently matched with

\*

\* \retval 0: no match

\* 1: match

\*/

static int DetectOpcuaMatch(DetectEngineThreadCtx \*det\_ctx, Packet \*p,

const Signature \*s, const SigMatchCtx \*ctx)

{

uint8\_t\* payload = p->payload;

uint16\_t payload\_len = p->payload\_len;

DetectOpcua\* opcua = (DetectOpcua\*)ctx;

if (payload\_len < OPCUA\_MIN\_FRAME\_LEN) {

SCLogNotice("payload length is too small");

return 0;

}

if (PKT\_IS\_PSEUDOPKT(p)) {

SCLogNotice("Pseudopkt detect");

return 0;

}

if (!PKT\_IS\_TCP(p)) {

SCLogNotice("Transport protocol does not TCP");

return 0;

}

uint8\_t type = \*(payload);

if (opcua->type && opcua->type != type) {

SCLogNotice("Packet does not pass the filtering by type, actual type = %d, rule = %d", type, opcua->type);

return 0;

}

uint8\_t function = \*(payload + OPCUA\_FUNC\_OFFSET);

if (opcua->function && opcua->function != function){

SCLogNotice("Packet does not pass the filtering by function, actual function = %d, rule = %d", function, opcua->function);

return 0;

}

if (opcua->size){

uint8\_t size = \*(payload + OPCUA\_SIZE\_OFFSET);

if (opcua->compare == LESS && opcua->size <= size){

SCLogNotice("Packet does not pass the filtering by size, actual size = %d, rule = %d", size, opcua->size);

return 0;

}

if (opcua->compare == EQUAL && opcua->size != size){

SCLogNotice("Packet does not pass the filtering by size, actual size = %d, rule = %d", size, opcua->size);

return 0;

}

if (opcua->compare == GREATER && opcua->size >= size){

SCLogNotice("Packet does not pass the filtering by size, actual size = %d, rule = %d", size, opcua->size);

return 0;

}

}

if (opcua->prev\_token){

uint8\_t token = \*(payload + OPCUA\_TOKEN\_OFFSET);

if (opcua->prev\_token == token){

SCLogNotice("Packet does not pass the filtering. Token is the same");

return 0;

}

else

opcua->prev\_token = token;

}

uint8\_t req = \*(payload + OPCUA\_REQ\_OFFSET);

if (opcua->req && opcua->req != req){

SCLogNotice("Packet does not pass the filtering by request id, actual request id = %d, rule = %d", req, opcua->req);

return 0;

}

SCLogNotice("PACKET PASSED the filtering, DETECT");

return 1;

}

/\*\* \internal

\*

\* \brief this function is used to add the parsed option into the current signature

\*

\* \param de\_ctx Pointer to the Detection Engine Context

\* \param s Pointer to the Current Signature

\* \param str Pointer to the user provided option

\*

\* \retval 0: Success

\* 1: Failure

\*/

static int DetectOPCUAopcuaSetup(DetectEngineCtx \*de\_ctx, Signature \*s,

const char \*str)

{

/\* store list id. Content, pcre, etc will be added to the list at this

\* id. \*/

s->init\_data->list = g\_opcua\_opcuabuf\_id;

DetectOpcua \*opcua = NULL;

SigMatch \*sm = NULL;

/\* set the app proto for this signature. This means it will only be

\* evaluated against flows that are ALPROTO\_OPCUA \*/

if (DetectSignatureSetAppProto(s, ALPROTO\_OPCUA) != 0)

SCReturnInt(-1);

if ((opcua = DetectOpcuaTypeParse(de\_ctx, str)) == NULL) {

SCLogNotice("type OPCUA NULL");

if ((opcua = DetectOpcuaSizeParse(de\_ctx, str)) == NULL){

SCLogNotice("size OPCUA NULL");

if ((opcua = DetectOpcuaTokenParse(de\_ctx, str)) == NULL){

SCLogNotice("token OPCUA NULL");

if ((opcua = DetectOpcuaReqParse(de\_ctx, str)) == NULL){

SCLogNotice("request id OPCUA NULL");

if ((opcua = DetectOpcuaFunctionParse(de\_ctx, str)) == NULL) {

SCLogNotice("function OPCUA NULL");

SCLogError(SC\_ERR\_PCRE\_MATCH, "invalid opcua option ");

goto error;

}

}

}

}

}

/\* Lets get this into a SigMatch and put it in the Signature. \*/

sm = SigMatchAlloc();

if (sm == NULL)

goto error;

sm->type = DETECT\_AL\_OPCUA\_OPCUABUF;

sm->ctx = (void \*) opcua;

SigMatchAppendSMToList(s, sm, DETECT\_SM\_LIST\_MATCH);

SCReturnInt(0);

return 0;

error:

if (opcua != NULL)

DetectOpcuaFree(de\_ctx, opcua);

if (sm != NULL)

SCFree(sm);

SCReturnInt(-1);

}

/\*\*

\* \brief Registration function for OPCUA keyword

\*/

void DetectOPCUAopcuabufRegister(void)

{

sigmatch\_table[DETECT\_AL\_OPCUA\_OPCUABUF].name = "opcua";

sigmatch\_table[DETECT\_AL\_OPCUA\_OPCUABUF].desc = "OPCUA content modififier to match on the opcua buffers";

sigmatch\_table[DETECT\_AL\_OPCUA\_OPCUABUF].Setup = DetectOPCUAopcuaSetup;

sigmatch\_table[DETECT\_AL\_OPCUA\_OPCUABUF].Match = DetectOpcuaMatch;

sigmatch\_table[DETECT\_AL\_OPCUA\_OPCUABUF].Free = DetectOpcuaFree;

#ifdef UNITTESTS

sigmatch\_table[DETECT\_AL\_OPCUA\_OPCUABUF].RegisterTests =

DetectOPCUAopcuabufRegisterTests;

#endif

sigmatch\_table[DETECT\_AL\_OPCUA\_OPCUABUF].flags |= SIGMATCH\_NOOPT;

DetectSetupParseRegexes(PARSE\_REGEX\_FUNCTION, &function\_parse\_regex);

DetectSetupParseRegexes(PARSE\_REGEX\_TYPE, &type\_parse\_regex);

DetectSetupParseRegexes(PARSE\_REGEX\_SIZE, &size\_parse\_regex);

DetectSetupParseRegexes(PARSE\_REGEX\_TOKEN, &token\_parse\_regex);

DetectSetupParseRegexes(PARSE\_REGEX\_REQ, &req\_parse\_regex);

SCLogNotice("OPCUA application layer detect registered.");

}

#ifdef UNITTESTS

#include "tests/detect-opcua-opcuabuf.c"

#endif