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.

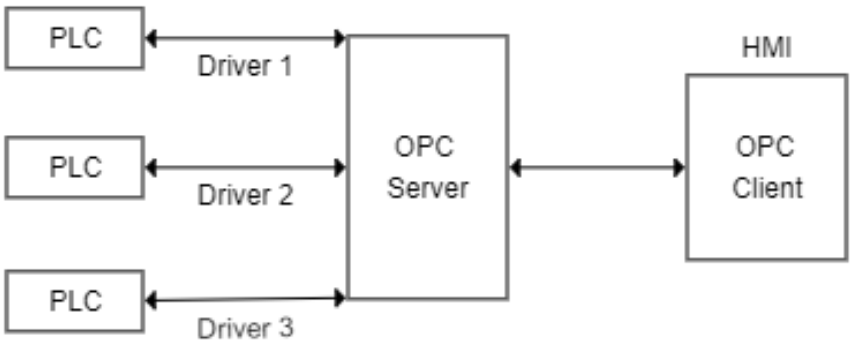
СОДЕРЖАНИЕ

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.

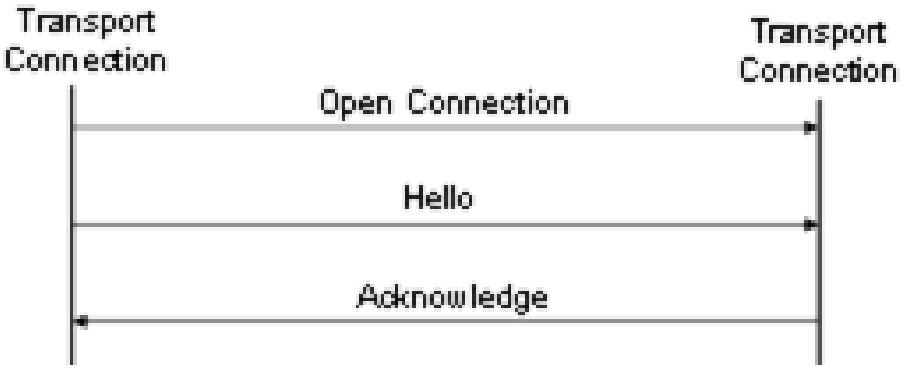
1.1. Кодировки

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.

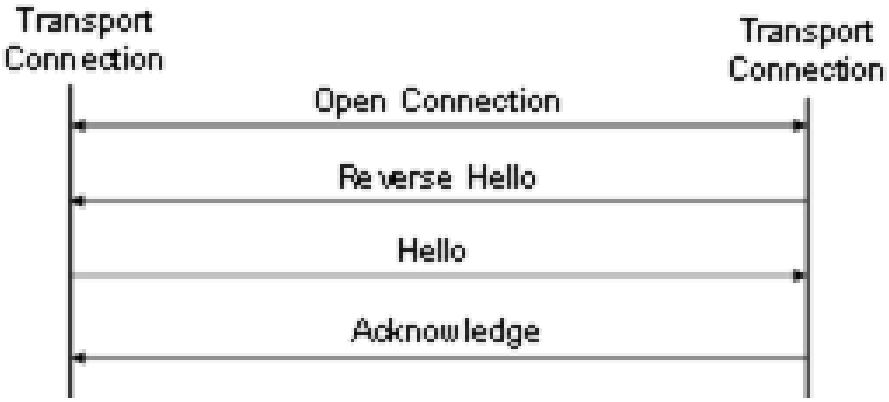
1.2. Взаимодействие между клиентом и сервером.

Для начала взаимодействия между клиентом и сервером необходимо установить соединение:

1. Соединение, инициированное клиентом



2. 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.

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

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

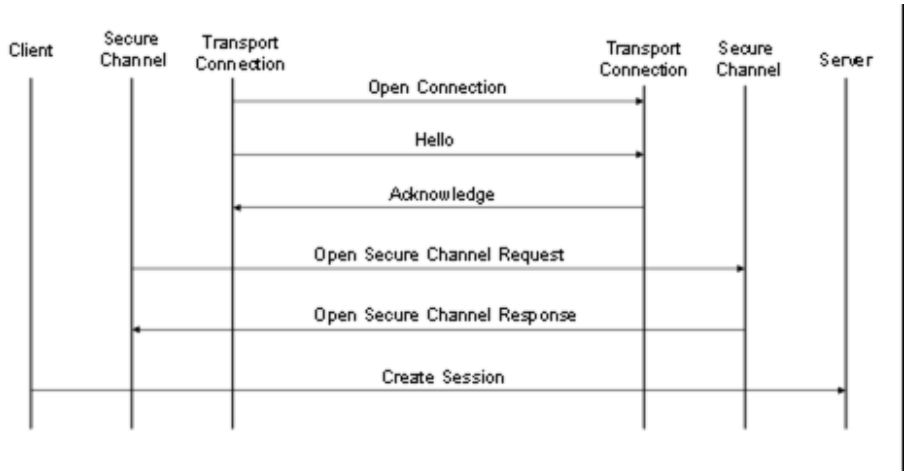
3. **Acknowlege Message** – сообщение, которое сервер посылает клиенту в ответ на сообщение Hello.

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.

1.3. 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.
requestedLifetim	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*

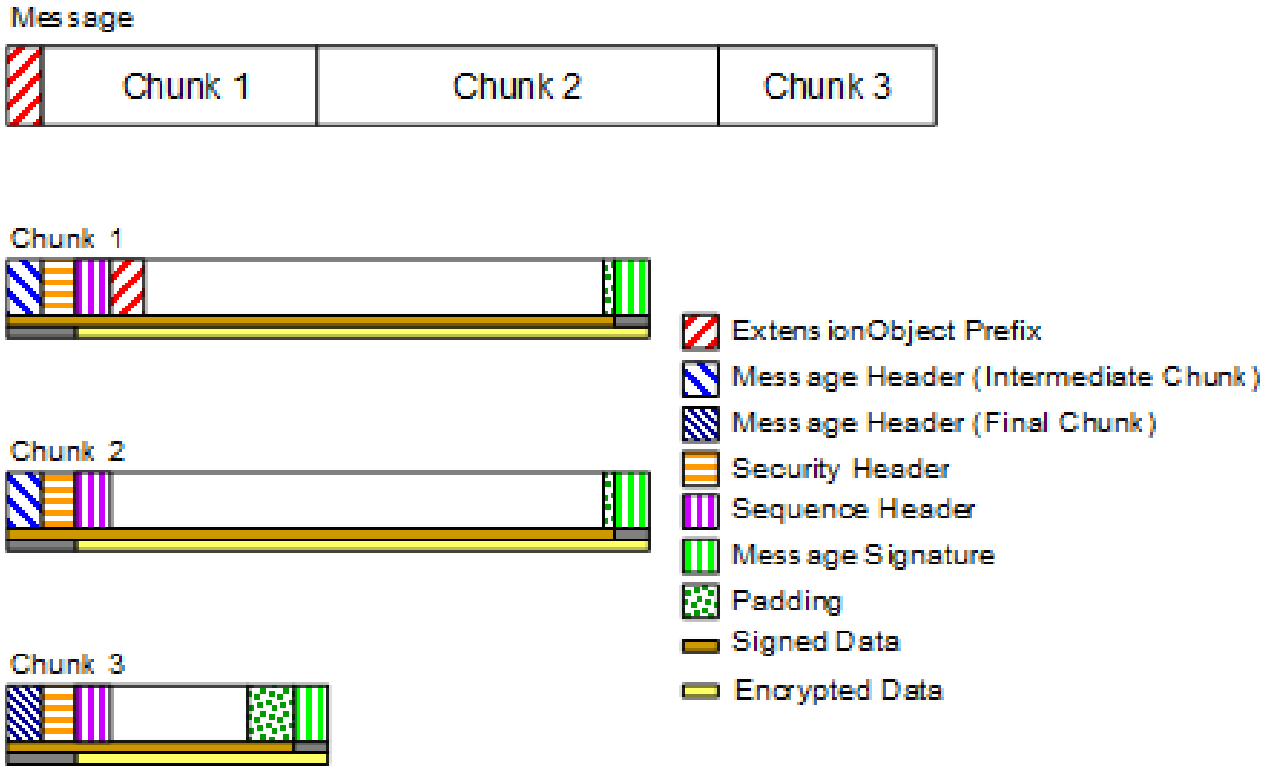
tokenId	ByteString	Уникальный идентификатор для конкретного *SecurityToken*
createdAt	UtcTime	Время создания *SecurityToken*
revisedLifetime	Duration	Время жизни *SecurityToken* в миллисекундах
serverNonce	ByteString	Случайное число, которое не должно использоваться ни в каком другом запросе

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.

1.4. 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	Типы	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.

2. **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
ReceiverCertificateThumbprint Length	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.

3. Sequence header – заголовок гарантирует, 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.

1.5. 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

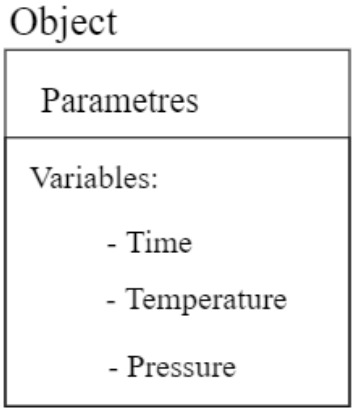
2. Описание лабораторного стенда

2.1. 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.

2.2. 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.

3. 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.

3.1. 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

3.2. 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.

3.3. 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;)

2. Детектирование запроса на чтение параметров.

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

opcua: function readReq; sid:2;)

3. Детектирование открытия нового канала.

alert tcp any any -> any any (msg: "OPN message"; opcua: type OPN;
classtype: bad-unknown; sid:3;)

4. Детектирование сообщений HELLO.

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

5. Удаление пакетов, размер которых меньше100.

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

6. Выявление пакетов, размер которых равен 171.

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

7. Удаление пакетов, токен безопасности которых недопустим.

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

8. Выявление запросов, id которых равен 29.

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

9. Выявление запроса на чтение параметров.

alert tcp any any -> any any (msg: "Read req"; opcua: function readReq; classtype: successful-recon-largescale; sid:9;)

10. 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.

4. Запуск ПО Suricata.

1. Модификация конфигурационного файла **/etc/suricata/suricata.yaml**:

a. Изменение адреса сети, в которой необходимо детектировать трафик.
b. Изменить интерфейс, через который будет проходить трафик.
c. Изменить путь к файлам с правилами для детектирования пакетов.

2. Описание правил для обнаружения пакетов.

- Формат файла - **.rules**
- Он должен находиться в директории, указанной в конфигурационном файле.
- Должен быть прописан сам в конфигурационном файле

3. Запуск ПО Suricata:

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

```
kali@kali:~$ sudo suricata -c /etc/suricata/suricata.yaml -i lo --init-errors-fatal
[5076] 13/12/2020 -- 06:32:19 - (suricata.c:1065) <Notice> (LogVersion) -- This is Suricata version 6.0.1-dev (75c0
f9bd0 2020-11-19) running in SYSTEM mode
[5076] 13/12/2020 -- 06:32:19 - (app-layer-opcua.c:487) <Notice> (RegisterOPCUAParsers) -- OPCUA TCP protocol detec
tion enabled.
[5076] 13/12/2020 -- 06:32:19 - (app-layer-opcua.c:504) <Notice> (RegisterOPCUAParsers) -- No opcua app-layer confi
guration, enabling echo detection TCP detection on port 4840.
[5076] 13/12/2020 -- 06:32:19 - (app-layer-opcua.c:524) <Notice> (RegisterOPCUAParsers) -- Registering OPCUA protoc
ol parser.
[5076] 13/12/2020 -- 06:32:19 - (detect-opcua-opcuabuf.c:543) <Notice> (DetectOPCUAopcuabufRegister) -- OPCUA appli
cation layer detect registered.
[5076] 13/12/2020 -- 06:32:19 - (output-json-opcua.c:195) <Notice> (JsonOPCUALogRegister) -- OPCUA JSON logger regi
stered.
[5076] 13/12/2020 -- 06:32:19 - (output-json-opcua.c:131) <Notice> (OutputOPCUALogInitSub) -- OPCUA log sub-module
initialized.
[5076] 13/12/2020 -- 06:32:19 - (detect-opcua-opcuabuf.c:362) <Notice> (DetectOpcuaReqParse) -- Request id: 29
[5076] 13/12/2020 -- 06:32:19 - (util-ioctl.c:322) <Warning> (SetEthtoolValue) -- [ERRCODE: SC_ERR_SYSCALL(50)] - F
ailure when trying to set feature via ioctl for 'lo': Operation not supported (95)
[5076] 13/12/2020 -- 06:32:19 - (tm-threads.c:1964) <Notice> (TmThreadWaitOnThreadInit) -- all 2 packet processing
threads, 4 management threads initialized, engine started.
```

- **c** - путь к конфигурационному файлу
- **i** - название интерфейса
-- **init-errors-fatal** - при обнаружении ошибки Suricata завершает работу

4. Запуск сервера.

python3 OPCUA_server.py

```
kali@kali:~/Desktop$ python3 OPCUA_server.py
Endpoints other than open requested but private key and certificate are not set.
Listening on 127.0.0.100:4840
Server started at opc.tcp://127.0.0.100:4840
39 789 2020-12-13 06:33:28.687612
30 582 2020-12-13 06:33:30.688055
22 943 2020-12-13 06:33:32.691128
```

5. Запуск клиента.

python3 OPCUA_client.py

```
kali@kali:~/Desktop$ python3 OPCUA_client.py
Client connected
Root node is:  i=84
Objects node is:  i=85
Children of root are:  [Node(NumericNodeId(i=85)), Node(NumericNodeId(i=86)), Node(NumericNodeId(i=87))]
Time:  2020-12-13 06:33:58.710300
       Pressure:  222
       Temperature:  39
Time:  2020-12-13 06:34:00.713045
       Pressure:  426
       Temperature:  10
Time:  2020-12-13 06:34:02.714520
       Pressure:  876
       Temperature:  44
```

6. Проверка детектирования пакетов по созданным правилам.

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

```
kali@kali:~/Desktop/suricata$ sudo tail -f 10 /var/log/suricata/fast.log
tail: cannot open '10' for reading: No such file or directory
==> /var/log/suricata/fast.log <==
12/13/2020-06:37:39.408228  [wDrop] [**] [1:10:0] MSG message type [**] [Classification: Not Suspicious Traffic] [Priority: 3]
 127.0.0.1:52444 → 127.0.0.100:4840
12/13/2020-06:37:39.408228  [**] [1:9:0] Read req [**] [Classification: Large Scale Information Leak] [Priority: 2] {TCP} 127.0
52444 → 127.0.0.100:4840
12/13/2020-06:37:39.408228  [**] [1:2:0] Request to read parameters [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1
4 → 127.0.0.100:4840
12/13/2020-06:37:39.408228  [**] [1:6:0] Packet size eq 171 [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:52444 -
.0.0.100:4840
12/13/2020-06:37:39.409307  [wDrop] [**] [1:5:0] Packet size lt 100 [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1
40 → 127.0.0.1:52444
12/13/2020-06:37:39.409307  [wDrop] [**] [1:10:0] MSG message type [**] [Classification: Not Suspicious Traffic] [Priority: 3]
 127.0.0.100:4840 → 127.0.0.1:52444
12/13/2020-06:37:39.409307  [**] [1:6:0] Packet size eq 171 [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.100:4840
7.0.0.1:52444
12/13/2020-06:37:39.409311  [wDrop] [**] [1:5:0] Packet size lt 100 [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1
40 → 127.0.0.1:52444
12/13/2020-06:37:39.409311  [wDrop] [**] [1:10:0] MSG message type [**] [Classification: Not Suspicious Traffic] [Priority: 3]
 127.0.0.100:4840 → 127.0.0.1:52444
12/13/2020-06:37:39.409311  [**] [1:6:0] Packet size eq 171 [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.100:4840
7.0.0.1:52444
```

- **f –** просмотр последних 10 строк в режиме реального времени.

ПРИЛОЖЕНИЕ

A. 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")
```

B. 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")
```

C. 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__ */
```

D. 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
}
```

E. 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__ */
```

F. 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
modifier 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
```