MASTER OF COMPUTER ENGINEERING FOR ROBOTICS AND SMART INDUSTRY (LM32) AY 2020/2021

Project title: Protocol OPC_UA

Class: Cyber Security for IOT

Course instructor Prof.Mariano Ceccato

Authored by:

MONRUETHAI SUEKSAKAN
Marticola N.VR466571
Email : monruethai.sueksakan_03@studenti.univr.it
MUHAMMAD ANAS UDDIN
Marticola N.VR465960
Email : muhammadanas.uddin@studenti.univr.it

Abstract

In 2006 OPPC foundation release first specification for opc unified architecture protocol one of the industrial protocol that guarantee security features such as authentication,authorization,integrity and confidentiality,In this paper we are discussed about industrial iot (IIOT) based protocol OPC-UA and classical OPC which is the old version of OPC-UA based on microsoft DCOM technology ,we also discussed the available vulnerabilities in OPC-UA protocol and the types of attack which can be perform on OPC UA to compromise its security in the end we discussed the results with the final conclusion

Introduction

1. An OPC-UA stands for open platform communication- united architecture machine to machine communication protocol for industrial automation developed by OPC foundation ,OPC UA is a standard interface to communicate between the numerous of data sources including devices on factory floor ,laboratory equipment .following are the distinguish characteristics for opc ua server
2. Based on a client server communication

3. Focus on communicating with industrial equipment and systems for data collection and control
4. cross-platform not tied to one operating system or programming language
5. Service oriented architecture (SOA)

Classical OPC

OPC classic based on microsoft windows technologies using COM/DCOM (dISTRIBUTED COMPONENT OBJECT MODEL) for communication between software components in a distributed client server network OPC Classic specifications provides separate specification for exchanging process data alarms and historical data,

The original OPC classic specification is OPC DA (data access0 which define interface between client and server applications to exchange process and manufacturing data ,other OPC classic specifications include OPC alarms and events (OPC AE) and OPC historical data access (OPC-HDA)

OPC DATA ACCESS

OPC-DA is an important protocol of OPC classic that's received data from out of the control system into the other system ,each information about specific data point contain some information about it such as timestamp that give us the information about exact time when the value was read and other is quality which give us the basic information if the data is valid or not .

OPC Alarms And Events

Unlike the DA protocol, alarms and events have no current value. It’s similar to a subscription-based service where clients receive every event. Events are not tagged, and have no name or quality attribute.

OPC Historical Data Access (OPC-HDA)

DA contains historical data and supports long record sets for one or more data points. It was designed to provide a unified way to extract and distribute historical data stored in SCADA or historian systems

OPC UA Communication Layer

OPC UA is build to be platform independent and communication is build into layers on top of the standard TCP/IP stack.Above the standard transport layers there are two layers that handle a session and established a secure channel between client and server the transport layer is made up of TCP/IP and top of that SSL,HTTP or HTTPS The Communication layer secure the communication channel not just that the data is corrupted but also it secure the authentication so that the end points can't be infiltrated and changed.This is based on X.509 certificates that have three parts to it and the first peer to peer trust needs to be manually done but after that the rest is taken care of securely.

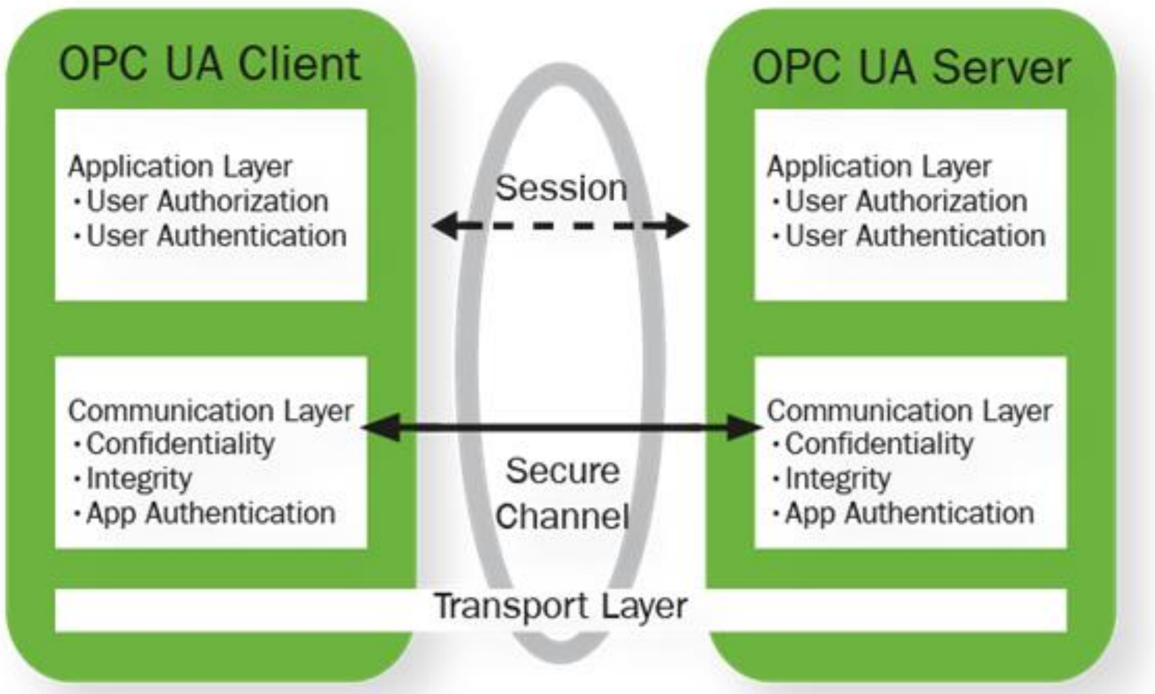


Figure A show OPC UA Communication Layers

OPC UA Security

As we know OPC UA IS developed for industrial components and interconnection of industrial components carried security concern that need to be addressed at different level to ensure secured industrial plant in order to established secure communication between industrial iot OPC UA network is used encryption and authentication features

There are six factors which describe secured industrial communication with OPC UA ,are following:

Security Mode

The OPC UA security mode set to sign,SignAndEncrypt,This ensures that authentication at the application level is forced. Security Mode SignAndEncrypt must be used if integrity and confidentiality of data has to be protected.

Selection Of Cryptographic Algorithm

At a minimum, the Security Policy 'Basic256Sha256' should be chosen if any existing client the server needs to interact with also supports this policy. Weaker security policies use outdated algorithms and should not be used. For example, SHA-1 is no longer secure and should not be used.

User Authentication

Its only for accessing non critical UA server resources as it does not provide any protection it is not possible to trace who is changed the data or configuration on the server side when this generic identifier was used an attacker can be used this identifier to read or write the data from server in unauthorized way if no restriction of the rights of the identifier found in that case anonymous was configured.

Certificate and private key storage

Never store private keys or the corresponding certificate files on an unencrypted file system. Use the dedicated certificate stores of your operating system and use operating system capabilities for setting the access rights.

Using certificates

Connections that do not provide trusted certificates should not be accepted, according to the white paper. The document further notes that "self-signed certificates should not be trusted automatically, which means without an additional verification. If the certificates are not self-signed, a Certificate Authority (CA), e.g., for all OPC UA applications of a company, is required.” This means that “the certificates of the CA are either self-signed or signed by another

Managing and maintaining certificates

Use certificate trust lists and certificate revocation lists to manage valid certificates. Only trusted users or processes should be allowed to write these lists. The lists should be updated regularly.

Vulnerabilities In OPC-UA

OPC-UA implement security measures at protocol level and like all opc protocols it is build on decoding and encoding of basic types, decoding and encoding flows are usually complex and provide good source of vulnerabilities

Softing Industrial Automation GmbH

In 2020, a team of researchers from Claroty analyzed products by several vendors that use Open Platform Communications protocols (OPC DA, AE, HDA, XML DA, DX и OPC UA). These protocols are used as third-party solutions by many large industrial automation system vendors, such as Rockwell Automation and GE. solutions based on these libraries are affected by numerous vulnerabilities, which could lead to equipment failure, remote code execution and leaks of critical data. According to Claroty, three software component vendors, Softing Industrial Automation GmbH, Kepware PTC, and Matrikon Honeywell, have released updates for their implementations of OPC functions, fixing stack and heap buffer overflow, use-after-free, improper exception handling, and uncontrolled resource consumption Softing Industrial Automation GmbH is a supplier of monitoring and diagnostic solutions for communication networks. The OPC Software Platform by Softing Industrial Automation is a solution that provides interoperability between OPC UA and OPC Classic, as well as implementing cloud connectivity. The integrated OPC UA Server provides access to data from PLCs manufactured by Siemens, Rockwell Automation, B&R Industrial Automation,

Mitsubishi, and other PLCs that communicate via the Modbus protocol.

The vulnerabilities identified do not require any special technical skills to exploit. They affect all Softing versions prior to 4.47.0

The first vulnerability (CVE-2020-14524) was identified in the OPC DA XML library of the Softing HTTP SOAP server. The web server does not limit header lengths and does not sanitize the values of SOAP headers. Using SOAP headers that are too large can eventually result in overconsumption of heap memory resources. Since the web server does not check the return code of the memory allocation, the data will be written to uninitialized memory, causing the web server to crash. This is a critical vulnerability with a CVSS v3.1 base score of 9.8

The second vulnerability (CVE-2020-14522) is a flaw that enables an invalid value to be used with certain parameters, creating an infinite memory allocation loop and ultimately causing high memory consumption resulting in denial-of-service conditions.vulnerabilities.

Kepware PTC

Kepware develops industrial connectivity software for enterprises in manufacturing, oil & gas, power & utilities, building automation, and other industries. Kepware solutions are used in SCADA systems, providing connectivity with Allen Bradley, AutomationDirect, BACnet, DNP 3.0, GE, Honeywell, Mitsubishi, Modicon, Omron, Siemens, Texas Instruments, Yokogawa and other industrial devices.

The following versions are vulnerable:

1. KEPServerEX v6.0 – v6.9;
2. ThingWorx Kepware Server v6.8 – v6.9;
3. ThingWorx Industrial Connectivity – all versions;
4. ThingWorx OPC-Aggregator – all versions.

A stack-based buffer overflow vulnerability (CVE-2020-27265) was identified in the ThingWorx Edge Server, which can be exploited remotely by unauthenticated attackers. A flaw in the logic for decoding OPC strings enables strings longer than 1024 bytes to be copied without allocating additional memory. By exploiting this vulnerability, attackers could overwrite data in the stack after the first 1024 bytes and cause the server to crash or, potentially, execute malicious code a CVSS v.3.1 base score of 9.8 has been calculated for this vulnerability

Matrikon Honeywell OPC UA Tunneler

The Matrikon OPC UA Tunneller enables client applications with OPC UA support to communicate not only with OPC UA servers, but also with the OPC Classic Server and clients.All Matrikon OPC UA Tunneller versions prior to 6.3.0.8233 are vulnerable.

Multiple vulnerabilities were found in Matrikon OPC UA Tunneller components, including a critical heap overflow flaw (CVE-2020-27297, CVSS v. 3.1 base score 9.8) and a memory leak due to a heap out of bounds read (CVE-2020-27299).

Types of OPC UA Encryption

Now that we have had a look at the functions OPC UA Certificates serve in the context of OPC UA security, we need to consider what happens to messages after you have trusted the OPC UA certificates and have enabled security on the OPC UA endpoint. Specifically, what does Sign&Encrypt mean on an endpoint and how can we be sure that the data is truly secure?
To really understand how OPC UA Security between two applications works we need to know a little about how symmetric and asymmetric (or public key) encryptions work – because at the end of the day they are not so different from one another. [1]

How Does Symmetric Encryption Work?

Symmetric encryption is so named (quite aptly) because the same key is used to both encrypt and decrypt the encoded message, not unlike a physical lock where the same key is used to lock and unlock the lock. This symmetry is great because it is a very fast way to encrypt/decrypt information because the same key is used for both, and because it is reasonably secure (although not as secure as its asymmetric brother).

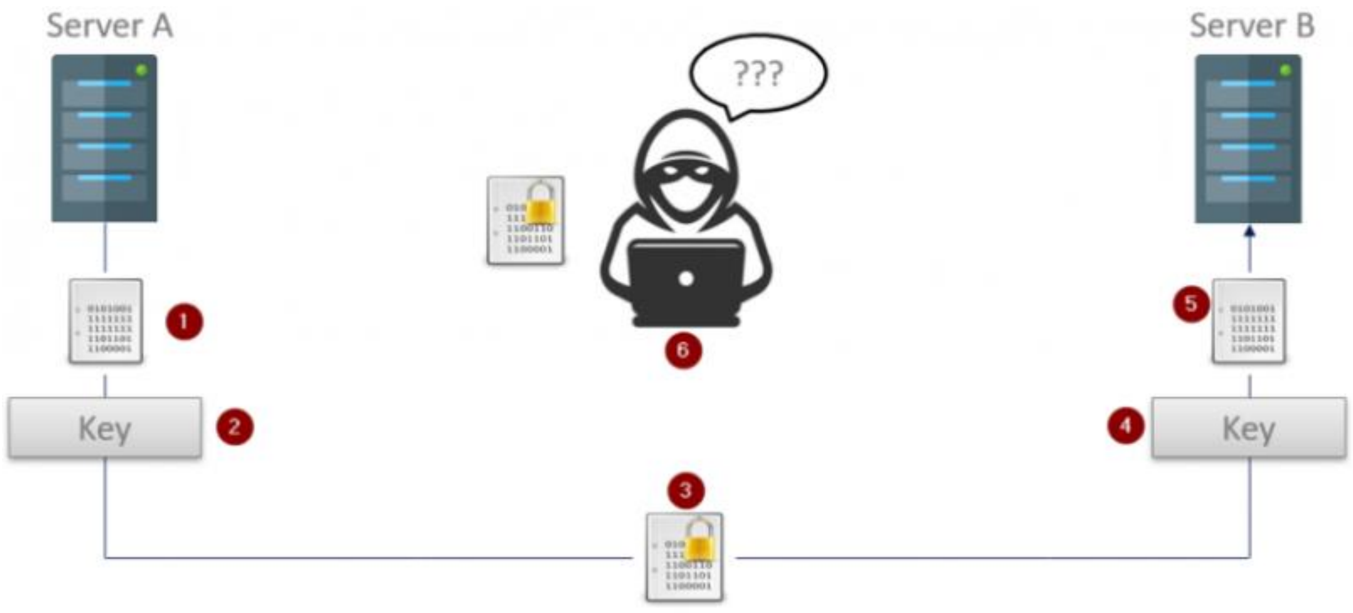


Figure B How Does Asymmetric Encryption Work

Asymmetric encryption – or public key encryption – (also quite aptly named) is considered asymmetric because the same key is no longer used for encryption and decryption. How is that possible, you ask? How can one key be used to encrypt data, and another key be used to decrypt it?
With asymmetric encryption, each application will have two certificates that are mathematically linked (more on what this means in a second):

1. A private certificate that is used for decryption and message signing
2. A public certificate that is used for encryption

These keys are mathematically linked meaning that a message that is encrypted with a public key can ONLY be decrypted with the corresponding private key.
The first step with any connection that will use asymmetric encryption is that the public keys are exchanged.
Because an encrypted message can only be decrypted using the corresponding private key, applications can be very liberal in who they give their public keys to since anyone that intercepts an encrypted message cannot decrypt the message (even if they have a copy of the public key used to do the encryption). This also means an application only needs a single public key that can be used by many applications – solving the certificate count issue we see when using symmetric encryption.

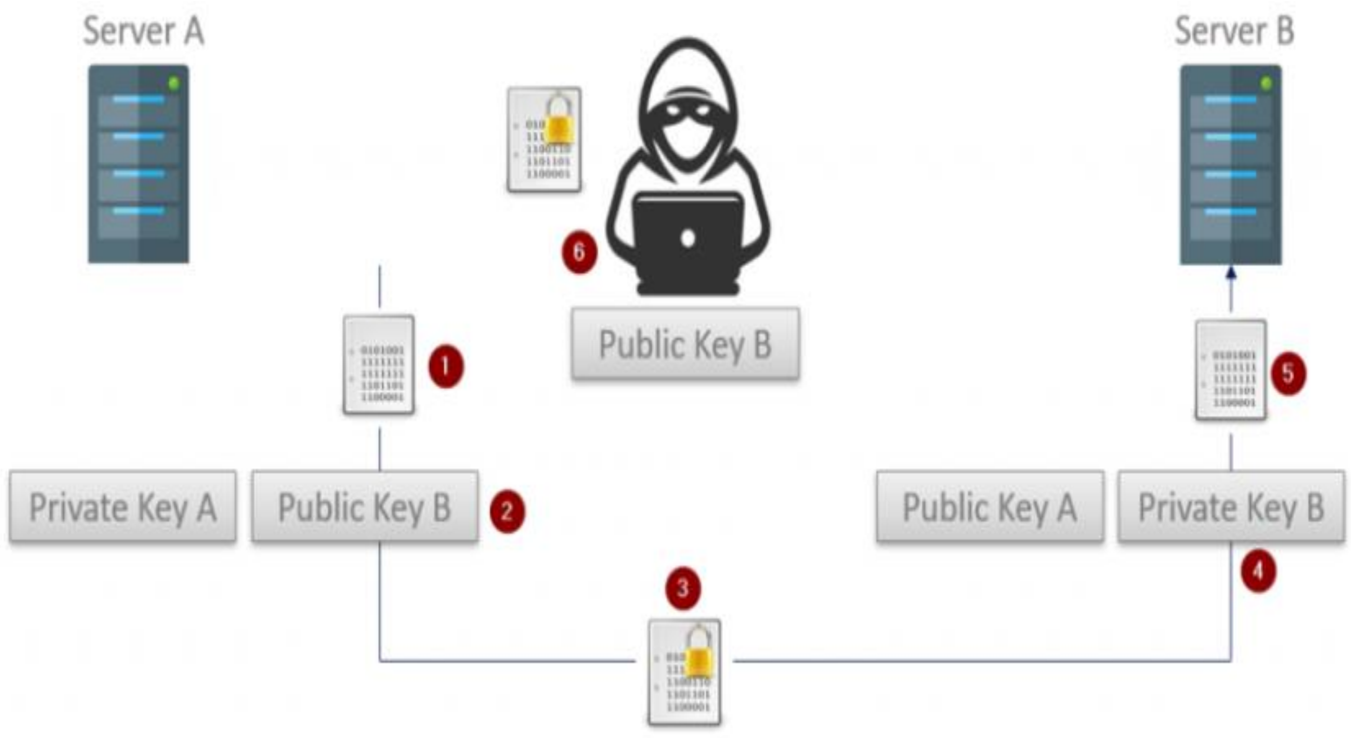


Figure C

1. The OPC UA Client running on Server A starts with an unencrypted message that will be sent to the OPC UA Server running on Server B. The public keys have already been exchanged.
2. The UA Client will use the public key that was received from the OPC UA Server to encrypt the messages and…
3. The encrypted message is sent off (where it is intercepted by a malicious third party)
4. The OPC UA Server on Server B uses its private key to decrypt the encoded message…
5. Which is then processed.
6. The bad actor that intercepted message as well as Server B’s public key is still unable to decrypt the captured message

OPC explain

What's OPA

Operational technology (OT) is a phrase used to describe equipment for controlling physical processes in the real world, whether passively recording information about a process or actively monitoring and/or changing the process. Industrial control systems (ICS) include all sorts of devices and protocols; however, one of their major components is the programmable logic controller (PLC). A PLC is the device responsible for the correct operation of field devices such as sensors, pumps, servomechanisms, and other devices

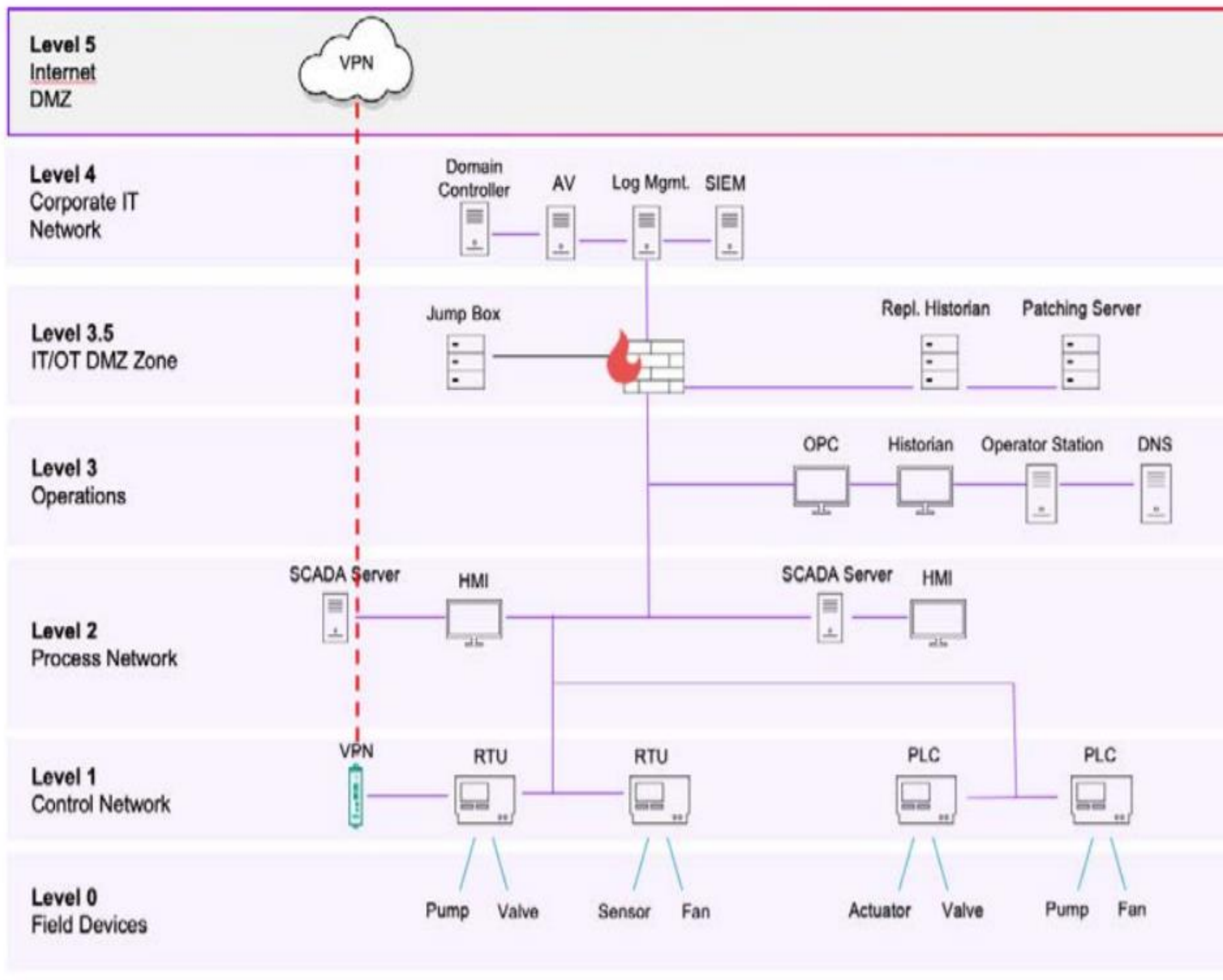


Figure D Model showing the componet of iot network

A lack of interoperability between ICS/SCADA protocols and the PLCs using them is a challenge. Each protocol is proprietary, and often products from different vendors are unable to exchange data. The development of the OPC protocol was meant to solve this problem. The idea was to create a standardized protocol for process control to allow easy maintenance from a single server which is capable of communicating with all endpoint devices in the OT network. The OPC Foundation defines OPC as the interoperability standard used for the secure and reliable exchange of data in the industrial automation environment and in other industries; the fact that OPC is platform independent allows devices from different vendors to seamlessly communicate

To achieve this goal, a few components were introduced: the OPC server, OPC client, and the OPC gateway. The OPC server contains many protocol converters that are capable of communicating with devices, such as PLCs, using their proprietary protocols, including Modbus, EtherNet/IP, S7, and more. The OPC server will constantly query devices for specific predefined memory values that are known as tags or points, and store them in a special database. Later, an OPC client, such as an HMI, can communicate with the server using the generic OPC protocol to get the values of these tags/points. Lastly, the OPC gateway is responsible for converting proprietary protocols into OPC

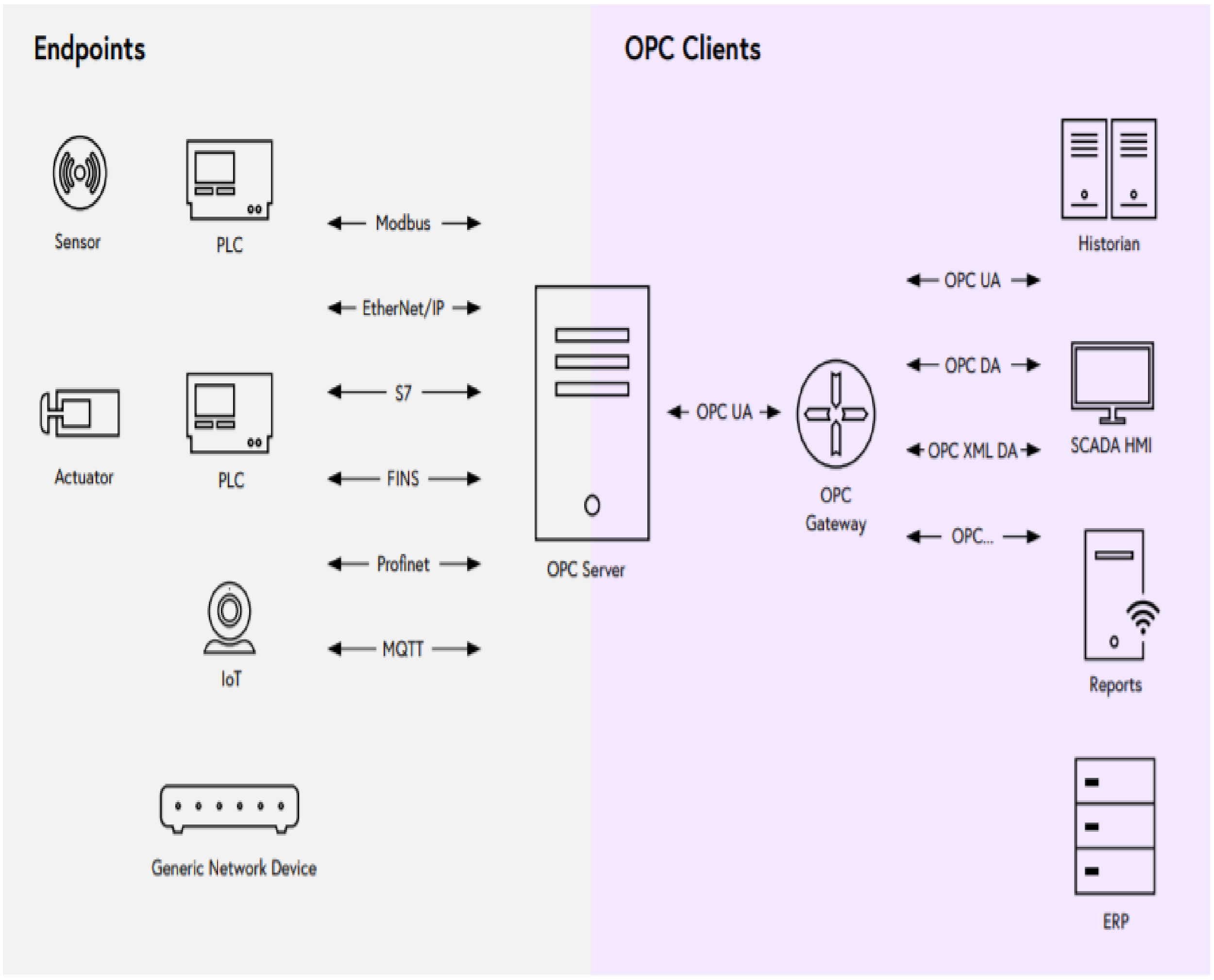


Figure E OPC servers convert proprietary protocols used by endpoint devices such as PLCs

GATEWAYS AND TUNNELLERS With the significant amount of sub-protocols and transport layers, a new component had to be introduced into the network architecture: the OPC Gateway, also known as an OPC Tunneller. The OPC Gateway allows for connections between different OPC servers and OPC clients regardless of the specific OPC protocol being used, vendor, version, or number of connections each OPC Server supports. For example, using an OPC gateway, you can connect any OPC DA server or client to any OPC UA server or client, locally or over the plant network.[Figture E]

Protocols in depth

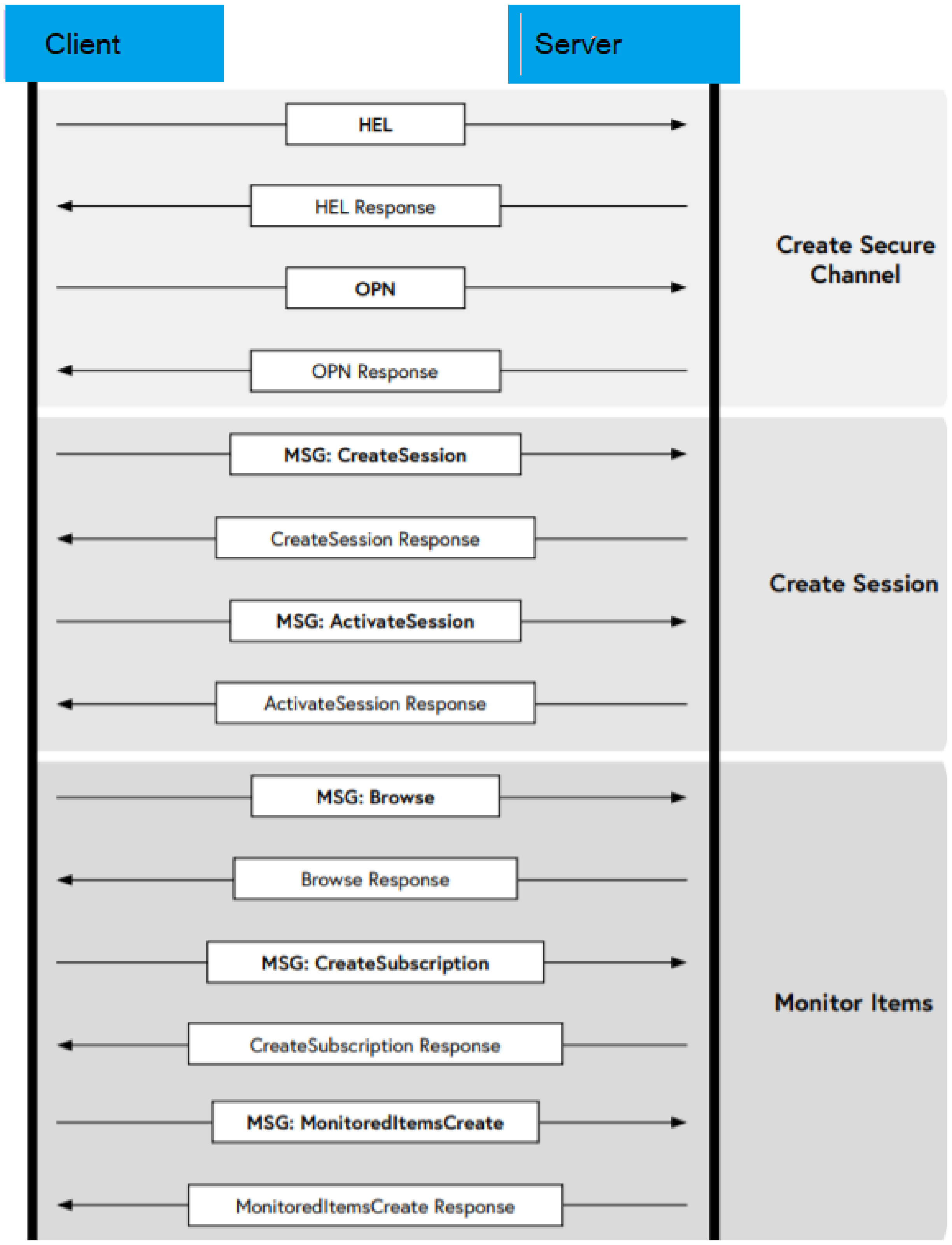


Figure F DCERPC communication initialization process

OPC DA
As the figure F have show, DCOM is used as the lower transport for the OPC DA service. This means that the initial connection is constructed above a DCERPC connection— the remote procedure call used in distributed computing environments—with OPC functionality. From a high-level perspective, the connection starts when the client initiates a DCOM connection, then the parties synchronize the mapped interfaces, which means each interface gets a unique identifier, and finally the client is able to use the functions exposed by the OPC interfaces, such as add group or item

The diagram, right, shows the basic connection initialization of DCERPC. After the TCP handshake, the client binds to the IOXIDResolver interface and calls the ServerAlive2 method. The server replies with a ServerAlive2 response which contains a string and security bindings (for example, the server's different IP addresses) and the client chooses the best settings that are compatible with both client and server. Finally the RemoteCreateInstance is called and the connection moves to a new high port for further OPC communications

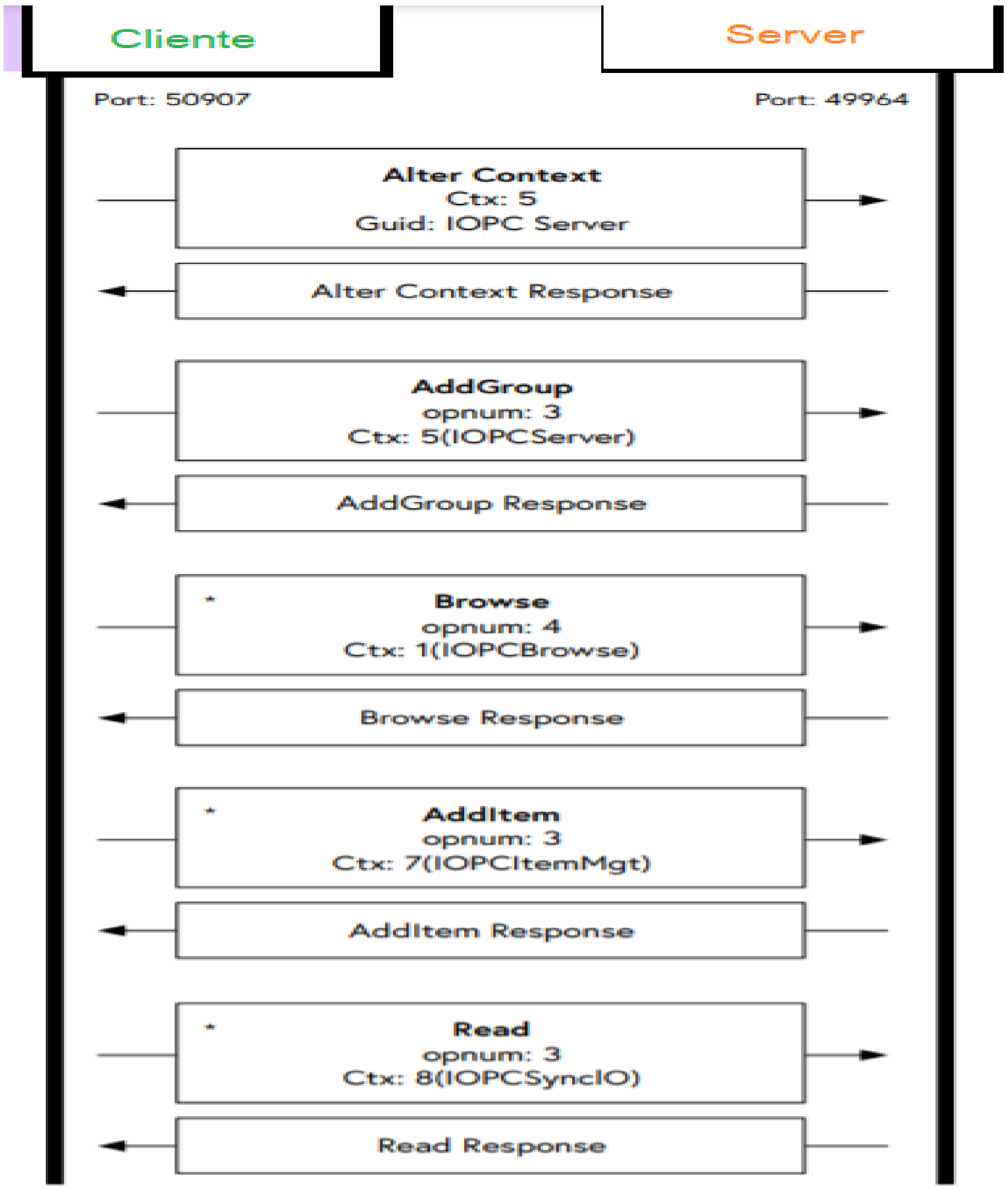


Figure F OPC DA example communication flow

The OPC protocol, like all COM-based protocols, has pre-built interfaces which describe the OPC functionality that can be called remotely. After the initial connection is made, the client maps these interfaces, which are identified by their UUID, to a Context ID using the AlterContext method. After the mapping is done the Context ID is used as the identifier for the interface. This means that to know what interface is called, one needs to also have the AlterContext packet sequence that mapped the UUID to the interface.

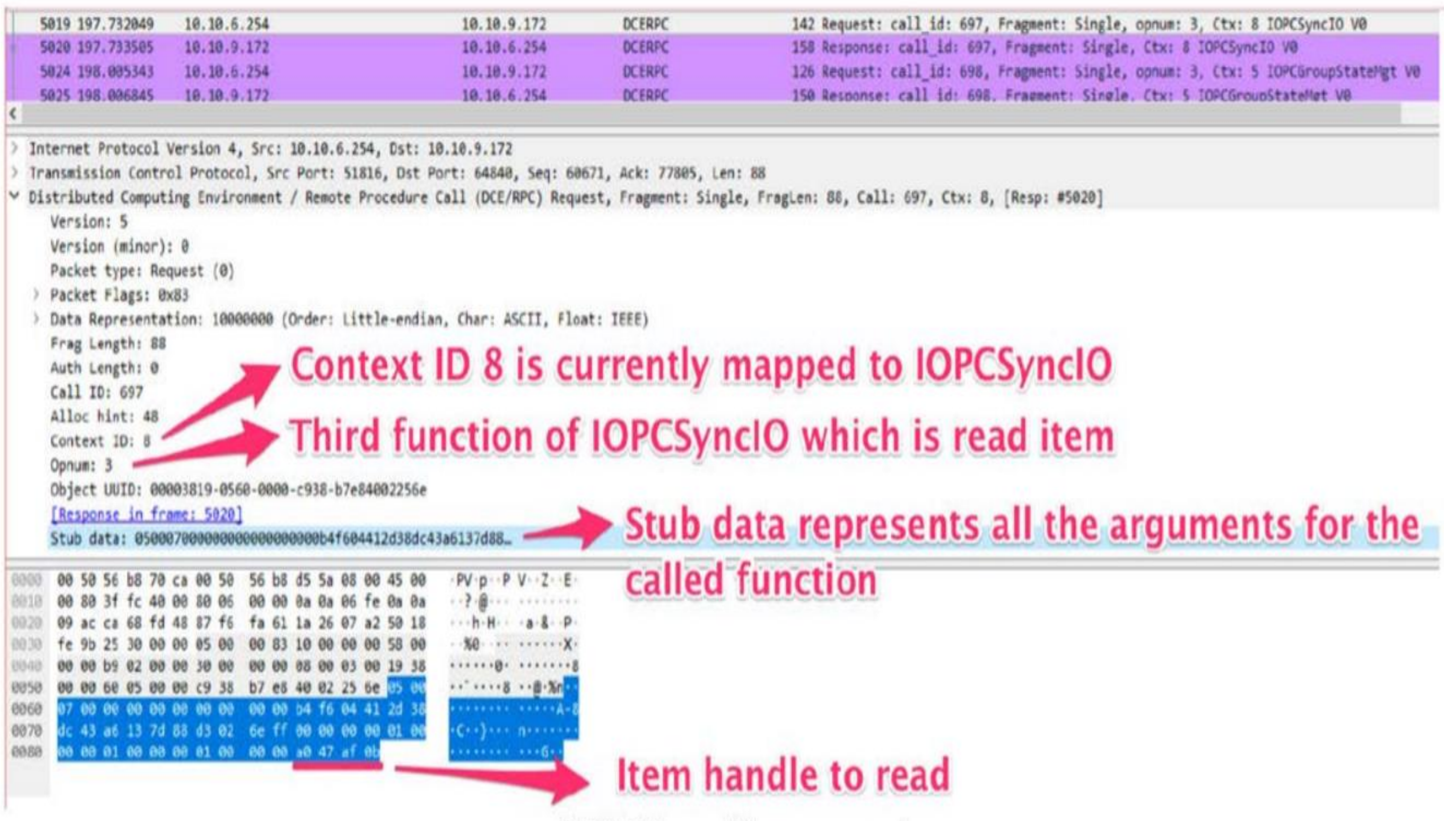
Each interface has predefined functions that are addressable by the opnum parameter. As with every COM object which implements the IUnknown interface, the first three functions are always QueryInterface, AddRef, and Release so the actual function opnum begins right after the Release function

OPC Groups provide a mechanism to logically organize items. Groups allow you to control and manage multiple items at once. Each group also has its own custom parameters and reading mechanism. Items must be inside a group so adding a group is usually what you see after mapping the interfaces.

Items are data points that represent elements. For example, an item can represent the current temperature received from a temperature sensor. The client can interact with items using item handles which represent the connection to a data source in the server and have Value, Quality and Time Stamp attributes associated with them. Item handles are obtained by calling the AddItem method. The client first browses the

server items using the Browse method and obtains the itemID linked to each item. The client sends the AddItem command along with the itemID of the requested item and finally, the server returns a handle for that item

Given all this information, here is a typical OPC DA packet sequence to read an item from the OPC server:



```
5019 197.732049   10.10.6.254   10.10.9.172   DCERPC   142 Request: call_id: 697, Fragment: Single, opnum: 3, Ctx: 8 IOPCSyncIO V0
5020 197.733505   10.10.9.172   10.10.6.254   DCERPC   158 Response: call_id: 697, Fragment: Single, Ctx: 8 IOPCSyncIO V0
5024 198.005343   10.10.6.254   10.10.9.172   DCERPC   126 Request: call_id: 698, Fragment: Single, opnum: 3, Ctx: 5 IOPCGroupStateMgt V0
5025 198.006845   10.10.9.172   10.10.6.254   DCERPC   150 Response: call_id: 698, Fragment: Single, Ctx: 5 IOPCGroupStateMgt V0

> Internet Protocol Version 4, Src: 10.10.6.254, Dst: 10.10.9.172
> Transmission Control Protocol, Src Port: 51816, Dst Port: 64840, Seq: 60671, Ack: 77805, Len: 88
v Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Request, Fragment: Single, FragLen: 88, Call: 697, Ctx: 8, [Resp: #5020]
    Version: 5
    Version (minor): 0
    Packet type: Request (0)
  > Packet Flags: 0x83
  > Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
    Frag Length: 88
    Auth Length: 0
    Call ID: 697
    Alloc hint: 48
    Context ID: 8
    Opnum: 3
    Object UUID: 00003819-0560-0000-c938-b7e84002256e
    [Response in frame: 5020]
    Stub data: 0500070000000000000000000b4f604412d38dc43a6137d88…

0000  00 50 56 b8 70 ca 00 50  56 b8 d5 5a 08 00 45 00   ·PV·p··P V··Z··E·
0010  00 80 3f fc 40 00 80 06  00 00 0a 0a 06 fe 0a 0a   ··?·@··· ········
0020  09 ac ca 68 fd 48 87 f6  fa 61 1a 26 07 a2 50 18   ···h·H·· ·a·&··P·
0030  fe 9b 25 30 00 00 05 00  00 83 10 00 00 00 58 00   ··%0···· ······X·
0040  00 00 b9 02 00 00 30 00  00 00 08 00 03 00 19 38   ······0· ·······8
0050  00 00 60 05 00 00 c9 38  b7 e8 40 02 25 6e 05 00   ··`····8 ··@·%n··
0060  07 00 00 00 00 00 00 00  00 00 b4 f6 04 41 2d 38   ········ ·····A-8
0070  dc 43 a6 13 7d 88 d3 02  6e ff 00 00 00 00 01 00   ·C··}··· n·······
0080  00 00 01 00 00 00 01 00  00 00 a0 47 af 0b         ········ ···G··
```

Context ID 8 is currently mapped to IOPCSyncIO
Third function of IOPCSyncIO which is read item
Stub data represents all the arguments for the called function
Item handle to read

OPC DA read item request.

Figure H example of OPC read item request as we captured in Wireshark

OPC UA

Every OPC UA message starts with a three byte ASCII code that makes the message type. The four main message types are:

HEL: Hello message

OPN: OpenSecureChannel message

MSG: A generic message container (secured with the channel's keys)

CLO: CloseSecureChannel message

The first message sent is the HEL message. The HEL message contains basic parameters to initialize the connection, such as the URL that the client wished to connect to, and the maximum message size the client

expects to receive. The HEL message is typically followed by the OPN request

OPN opens the secure channel, and if all goes well, the server responds with the SecureChannelID. With the channel open, the client and server can send MSG type messages over the open channel. For example, a common message being used is the MonitoredItemsCreate which the client uses in order to track items. The session ends with the CLO message which terminates the connection [figure F]

OPC UA VULNERABILITIES

OPC UA implements security measures at the protocol level, and like all OPC protocols, it is built on decoding and encoding basic types. That's why we decided to focus most of our research on the items decoding/encoding flows of the server. Decoding/encoding flows are usually complex and provide a good source for vulnerabilities. To have the greatest impact, we decided to focus on the encoding/decoding process of data types found in the initial message exchange because it occurs before the authentication phase. Therefore, the exploitation of these vulnerabilities requires no prior knowledge of the server and can be triggered more easily

KEPWARE'S THINGWORX EDGE AND KEPSERVEREX SERVERS

KEPServerEX OPC server software allows users to connect, manage, monitor, and control diverse automation devices. Kepware also provides ThingWorx Kepware Edge which implements most of the KEPServerEX features for Linux-based environments. Claroty discovered multiple vulnerabilities affecting KEPServerEX and ThingWorx Kepware Edge that lead to denial-ofservice attacks, sensitive data leaks, and could also potentially lead to code execution. Kepware's OPC protocol stack is embedded as a third-party component in many products across different industries

CVE-2020-27265

KEPWARE THINGWORX EDGE SERVER: STACK OVERFLOW DUE TO INTEGER OVERFLOW

We discovered a stack overflow vulnerability in the ThingWorx Kepware Edge server that could allow an attacker to crash the server, and under certain conditions, remotely execute code.

The first two messages in the OPC UA session are **HEL** followed by OPN. The HEL and OPN message includes a trove of client information and settings, giving us a vast attack surface before any authentication stages which could assist in exploiting the server without any prior knowledge.

For example, the HEL message is constructed as follows:

OPC UA HEL Message

Beside the normal reading method where a client periodically reads data from the server, OPC UA provides a more efficient way of transferring data using subscriptions. In the subscription model, the client specifies which items to track and the server takes care of the monitoring of these items. The client will only be notified in case of a change in one of the monitored items. The client can subscribe to the change of data values, object events, or aggregated values calculated at a client-defined interval.

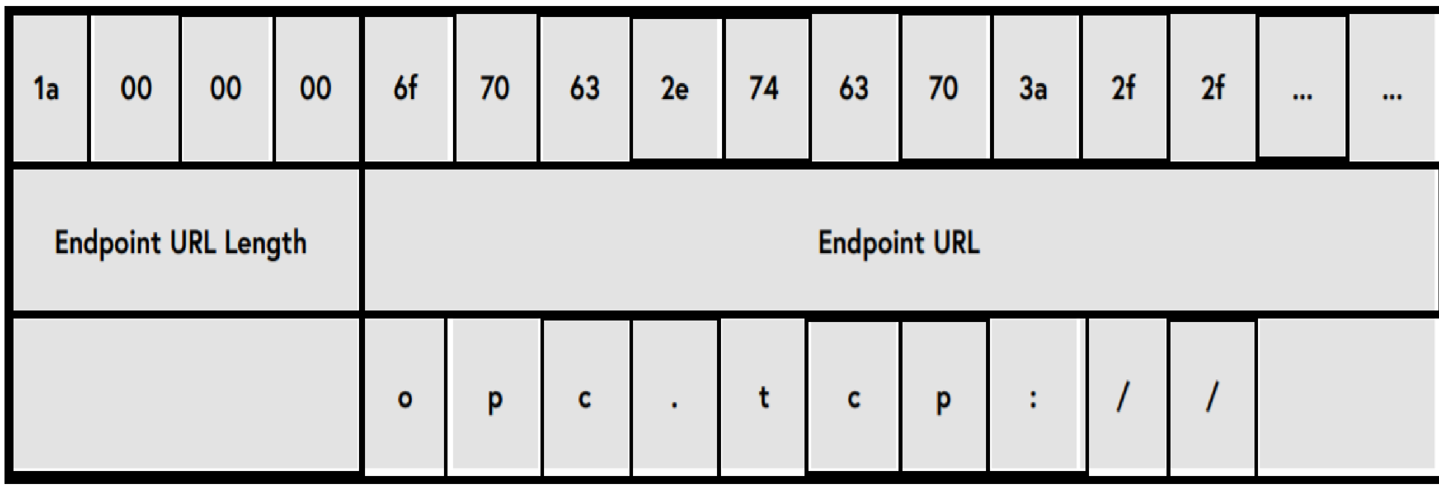
For example, the HEL message is constructed as follows:

Figure 3 example of an OPC UA packets sequence flow to monitor items' values

OPC UA HEL Message

48	45	4c	46	3a	00	00	00
Message Type			Chunk Type	Message Size			
M	S	G	F				

00	00	00	00	00	00	10	00	00	00	10	00	00	00	00	01	83	13	00	00
Protocol Version				Recieve Buffer Size				Send Buffer Size				Max Message Size				Max Chunk Count			



1a	00	00	00	6f	70	63	2e	74	63	70	3a	2f	2f	...	...
Endpoint URL Length				Endpoint URL											
				o	p	c	.	t	c	p	:	/	/		

The HEL message data is an OPC string of the endpoint URL. The OPC string contains two fields: string length and the string data. Both values are controlled by the client

We researched ThingWorx Kepware Edge and tried to understand how it decodes OPC strings. Finally we came to conclusion that this is the logic for OPC string decoding:

1) Read 4 bytes into the string_length variable

2) If string_length is less or equal to 0 return an empty string

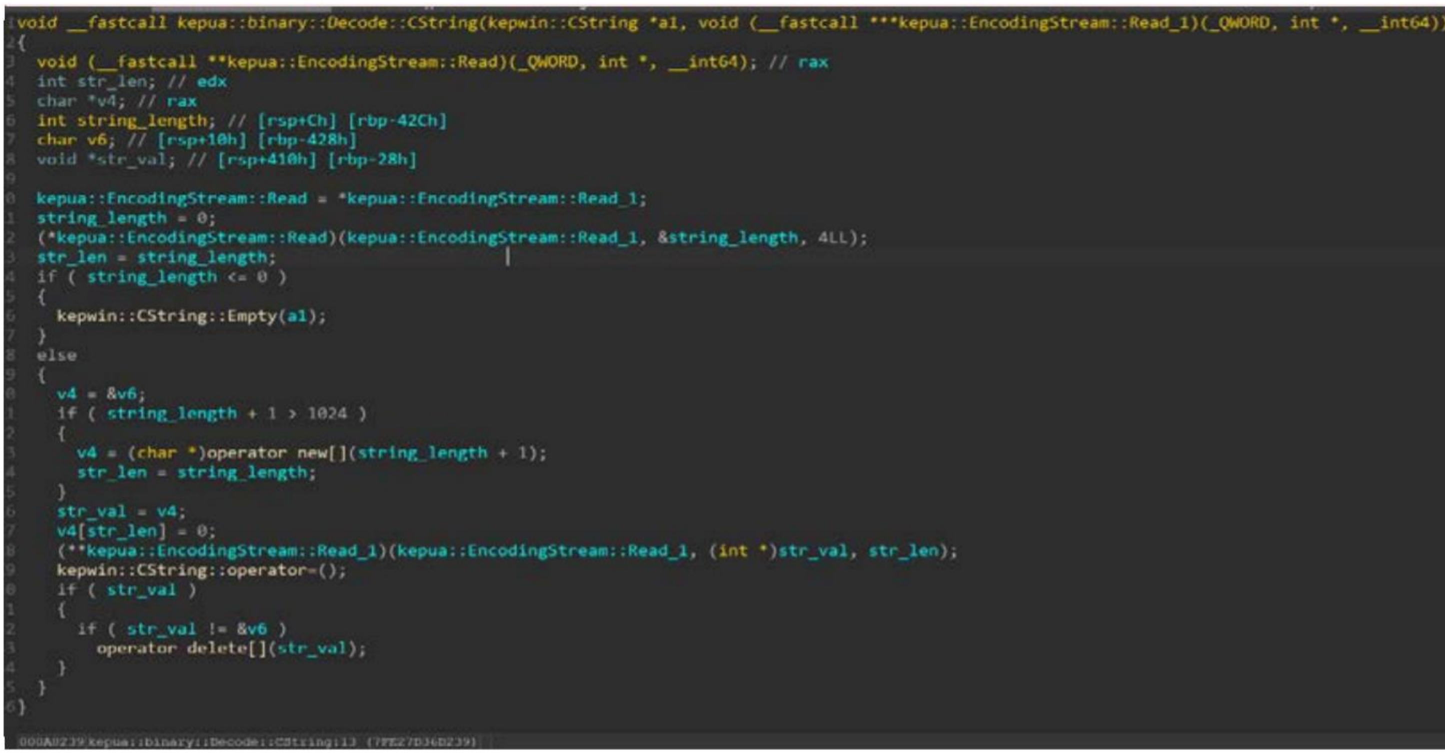
3) If string_length + 1 is more than 1024 bytes then allocate a new buffer with size of string_length +1

4) Add a null-terminator to the end of the buffer 5) Copy string_length bytes to ethe allocated buffer

In our research we found a flaw in this logic which makes it possible to copy a string bigger than 1024 without allocating more memory. To do that we need these two conditions to be false:

1) string_length <= 0

2) string_length + 1 > 1024



```
void __fastcall kepua::binary::Decode::CString(kepwin::CString *a1, void (__fastcall ***kepua::EncodingStream::Read_1)(_QWORD, int *, __int64))
{
  void (__fastcall **kepua::EncodingStream::Read)(_QWORD, int *, __int64); // rax
  int str_len; // edx
  char *v4; // rax
  int string_length; // [rsp+Ch] [rbp-42Ch]
  char v6; // [rsp+10h] [rbp-428h]
  void *str_val; // [rsp+410h] [rbp-28h]

  kepua::EncodingStream::Read = *kepua::EncodingStream::Read_1;
  string_length = 0;
  (*kepua::EncodingStream::Read)(kepua::EncodingStream::Read_1, &string_length, 4LL);
  str_len = string_length;
  if ( string_length <= 0 )
  {
    kepwin::CString::Empty(a1);
  }
  else
  {
    v4 = &v6;
    if ( string_length + 1 > 1024 )
    {
      v4 = (char *)operator new[](string_length + 1);
      str_len = string_length;
    }
    str_val = v4;
    v4[str_len] = 0;
    (**kepua::EncodingStream::Read_1)(kepua::EncodingStream::Read_1, (int *)str_val, str_len);
    kepwin::CString::operator=();
    if ( str_val )
    {
      if ( str_val != &v6 )
        operator delete[](str_val);
    }
  }
}
000A8239 kepua::binary::Decode::CString:13 (7FE27D36D239)
```

Figure 5 Kepware binary::Decode::CString function.

Fortunately the string_length parameter is a signed integer meaning that if the string length is 0x7FFFFFFF (max positive integer on 32 bit architecture) we will pass check No. 1 because the string_length is really bigger than 0 and also passes the second check because 0x7FFFFFFF + 1 overflows to a negative number (INT_MIN) and therefore is not bigger than 1024. We can use the HEL message and set the endpoint_url field (which is an OPC String) length to 0x7FFFFFFF to trigger this pre-authentication

The result is a stack buffer overflow because the allocated buffer is 1024 bytes on the stack but we can send a much larger buffer which will overwrite data on the stack after the first 1024 bytes

CVE-2020-27263

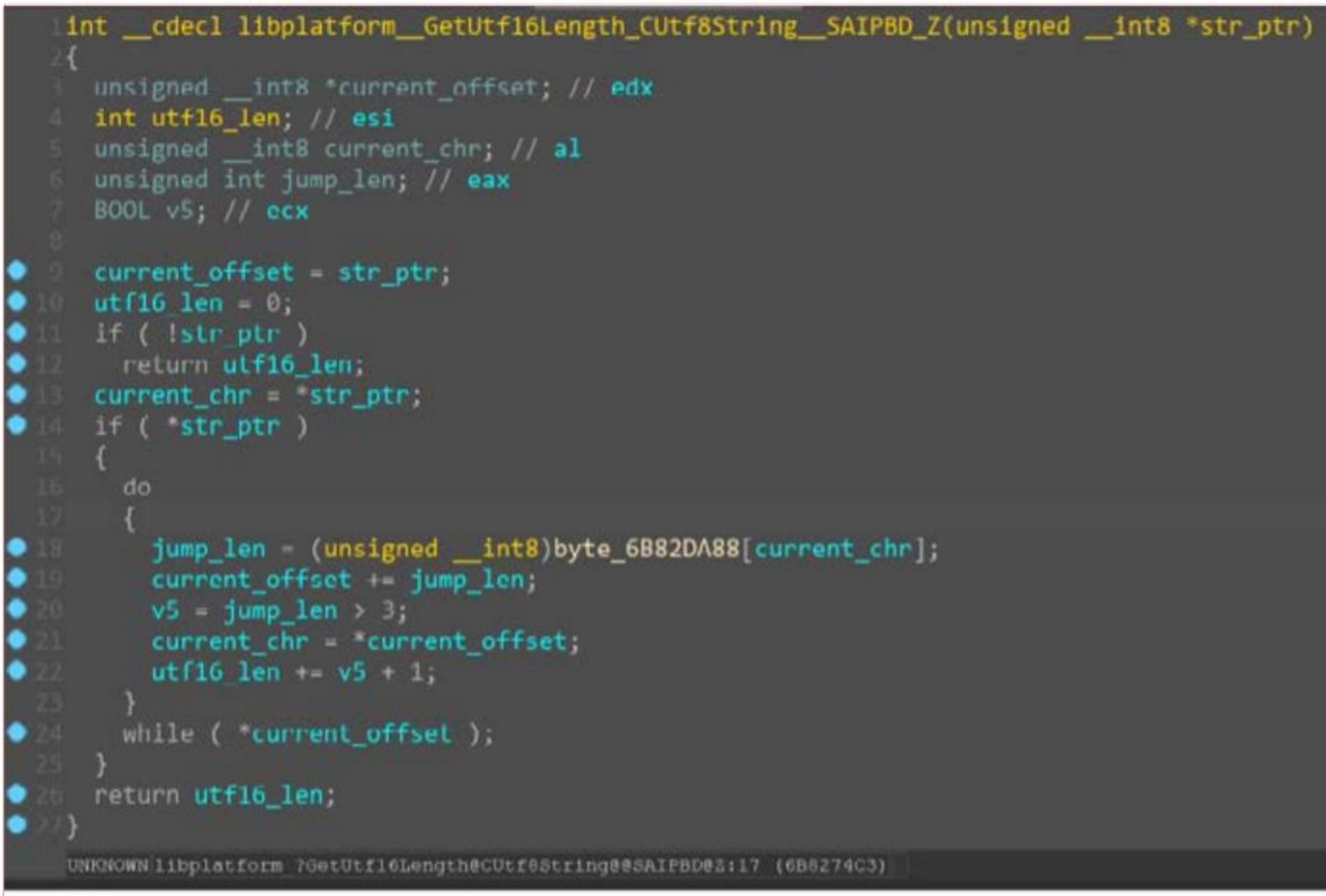
KEPWARE THINGWORX EDGE SERVER: INFORMATION LEAK DUE TO A HEAP OOB READ

The second Kepware vulnerability is also found in the string decoding flow, but can be accessed from other flows because it is a part of the libplatform library which is a shared cross platform utility library for all Kepware products. This affects Windows and Linux versions of the server, and could lead to information leaks and potential server crashes.

The CUtf8String::GetUtf16Length function returns the expected length of a utf-8 string formatted as utf-16 string. The function loops until a null-byte is found and in each iteration checks the length of the utf-8 char according to the conversion table, jumps that length, and adds 1 or 2 to the utf-16 total length accordingly. In our research, we found that the function fails to check that the current position is less or equal to the utf-8 string length, which leads to the reading of arbitrary memory values until a null terminator is reached

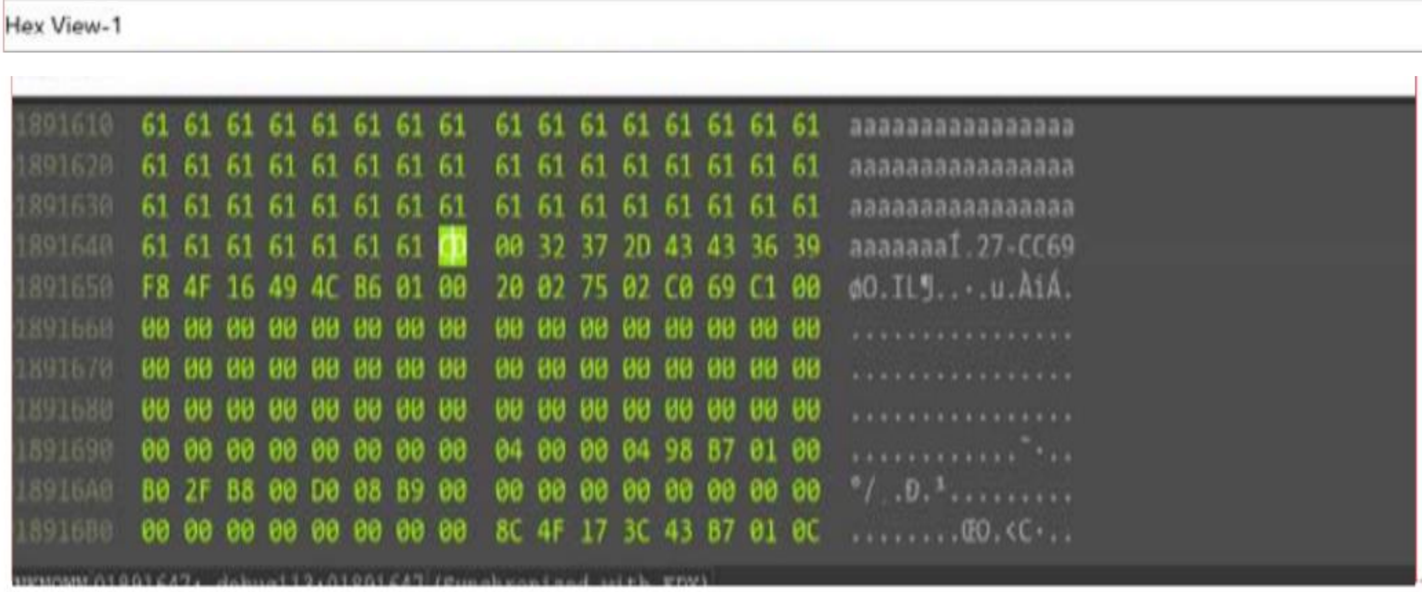
In the example below, we can see a string that consists of "a" chars (0x61), and the last byte is 0xcd. 0xcd is mapped to two bytes in the unicode mapping table, so instead of exiting the loop at the null-byte immediately after 0xcd, it will jump two bytes forward while "jumping" over the original string's null-terminator. Therefore, it will continue reading memory values from the heap until it hits another null-byte somewhere down the heap memory and after consuming large amounts of data.

The length returned from this is, of course, wrong and can lead to multiple logic errors down the line depending on where this is called from. For example,in one of the code flows we researched, the returned value was used in memcpy and so we were able to leak information from the heap



```
int __cdecl libplatform__GetUtf16Length_CUtf8String__SAIPBD_Z(unsigned __int8 *str_ptr)
{
  unsigned __int8 *current_offset; // edx
  int utf16_len; // esi
  unsigned __int8 current_chr; // al
  unsigned int jump_len; // eax
  BOOL v5; // ecx

  current_offset = str_ptr;
  utf16_len = 0;
  if ( !str_ptr )
    return utf16_len;
  current_chr = *str_ptr;
  if ( *str_ptr )
  {
    do
    {
      jump_len = (unsigned __int8)byte_6B82DA88[current_chr];
      current_offset += jump_len;
      v5 = jump_len > 3;
      current_chr = *current_offset;
      utf16_len += v5 + 1;
    }
    while ( *current_offset );
  }
  return utf16_len;
}
UNKNOWN libplatform_?GetUtf16Length@CUtf8String@@SAIPBD@Z:17 (6B8274C3)
```



Hex View-1

```
1891610  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
1891620  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
1891630  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
1891640  61 61 61 61 61 61 61 CD  00 32 37 2D 43 43 36 39  aaaaaaaÍ.27-CC69
1891650  F8 4F 16 49 4C B6 01 00  20 02 75 02 C0 69 C1 00  øO.IL¶..·.u.ÀiÁ.
1891660  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1891670  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1891680  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1891690  00 00 00 00 00 00 00 00  04 00 00 04 98 B7 01 00  ............˜·..
18916A0  B0 2F B8 00 D0 08 B9 00  00 00 00 00 00 00 00 00  °/¸.Ð.¹.........
18916B0  00 00 00 00 00 00 00 00  8C 4F 17 3C 43 B7 01 0C  ........ŒO.<C·..
```

figure 6 CUtf8String::GetUtf16Length function and memory values while debugging the function

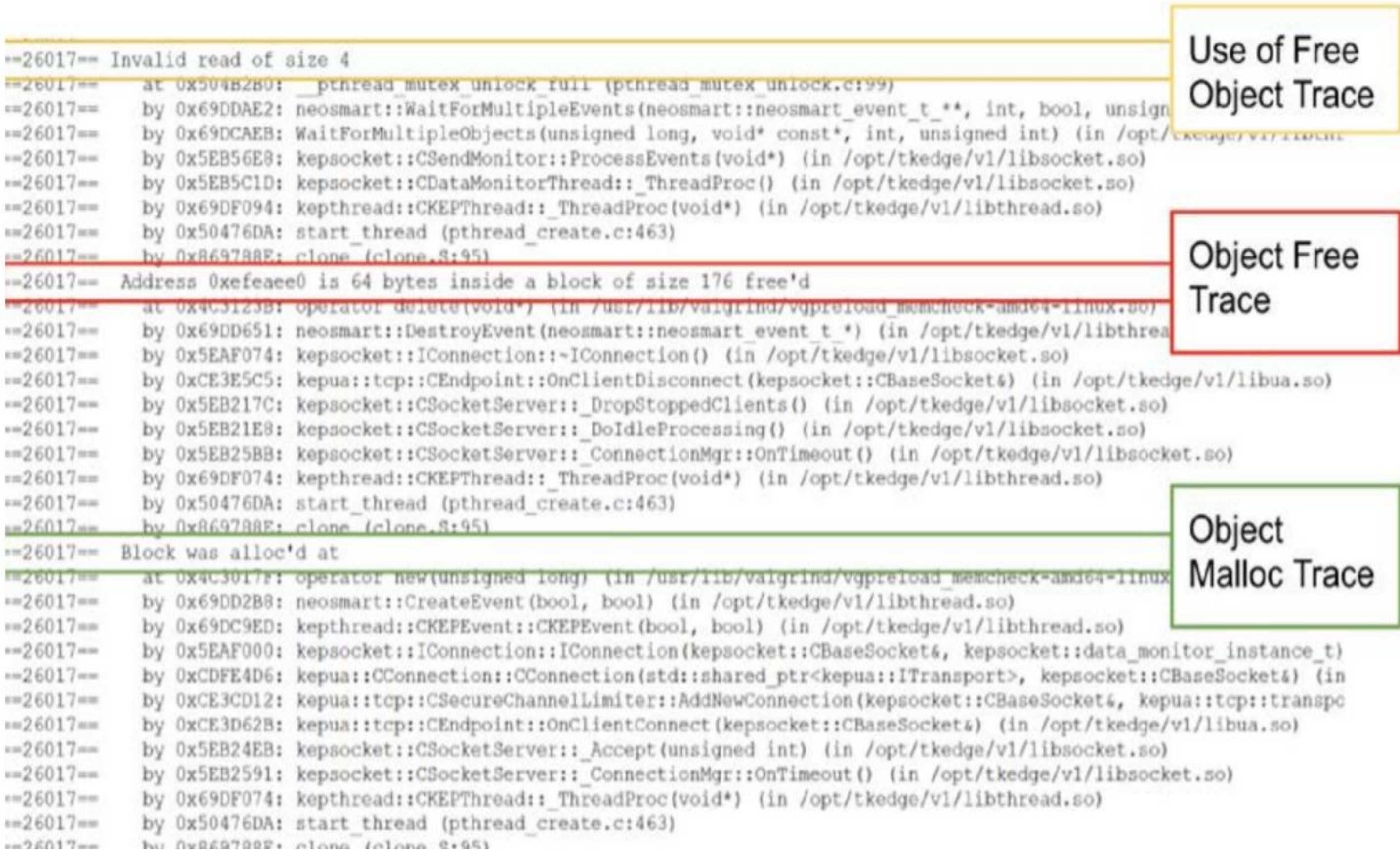
CVE-2020-27267

KEPWARE KEPSERVEREX EDGE SERVER: DENIAL OF SERVICE (DOS)DUE TO USE-AFTER-FREE (UAF)

Another good place for pre-authentication vulnerabilities is transport layers. For OPC DA, the COM DCOM transport is well-tested but other transport methods are usually up to the companies that write the server. Both OPC UA and OPC DA (in non-binary forms) require a well thought-out server that can handle multiple connections

We always like to have something running in the background when we start our research, so when we started the research on the Kepware products, the first thing we did was to set up a simple fuzzer. The fuzzer started with a seed of an OPC UA HEL and OPN requests and sent variations of them from 10 threads. This didn’t uncover anything interesting in the parsing flow, probably because every time we ran the fuzzer the server crashed after a few minutes

This crash was a bit harder to investigate, so we decided to run KEPServerEX with Valgrind. Valgrind is a suite of profiling and debugging features, with memcheck being the most popular one. Memcheck detects many memory-related errors by tracing malloc and free calls. Looking at the Valgrind log, we realized we have a race condition leading to a use-after-free condition. We investigated the trace and determined that an event for the connection is raised after the connection is closed and when the program tries to use the freed connection object it crashes



```
==26017== Invalid read of size 4
==26017==    at 0x504B2B0: __pthread_mutex_unlock_full (pthread_mutex_unlock.c:99)
==26017==    by 0x69DDAE2: neosmart::WaitForMultipleEvents(neosmart::neosmart_event_t_**, int, bool, unsign
==26017==    by 0x69DCAEB: WaitForMultipleObjects(unsigned long, void* const*, int, unsigned int) (in /opt/tkedge/v1/libth
==26017==    by 0x5EB56E8: kepsocket::CSendMonitor::ProcessEvents(void*) (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0x5EB5C1D: kepsocket::CDataMonitorThread::_ThreadProc() (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0x69DF094: kepthread::CKEPThread::_ThreadProc(void*) (in /opt/tkedge/v1/libthread.so)
==26017==    by 0x50476DA: start_thread (pthread_create.c:463)
==26017==    by 0x869788E: clone (clone.S:95)
==26017==  Address 0xefeaee0 is 64 bytes inside a block of size 176 free'd
==26017==    at 0x4C3123B: operator delete(void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==26017==    by 0x69DD651: neosmart::DestroyEvent(neosmart::neosmart_event_t_*) (in /opt/tkedge/v1/libthrea
==26017==    by 0x5EAF074: kepsocket::IConnection::~IConnection() (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0xCE3E5C5: kepua::tcp::CEndpoint::OnClientDisconnect(kepsocket::CBaseSocket&) (in /opt/tkedge/v1/libua.so)
==26017==    by 0x5EB217C: kepsocket::CSocketServer::_DropStoppedClients() (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0x5EB21E8: kepsocket::CSocketServer::_DoIdleProcessing() (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0x5EB25BB: kepsocket::CSocketServer::_ConnectionMgr::OnTimeout() (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0x69DF074: kepthread::CKEPThread::_ThreadProc(void*) (in /opt/tkedge/v1/libthread.so)
==26017==    by 0x50476DA: start_thread (pthread_create.c:463)
==26017==    by 0x869788E: clone (clone.S:95)
==26017==  Block was alloc'd at
==26017==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux
==26017==    by 0x69DD2B8: neosmart::CreateEvent(bool, bool) (in /opt/tkedge/v1/libthread.so)
==26017==    by 0x69DC9ED: kepthread::CKEPEvent::CKEPEvent(bool, bool) (in /opt/tkedge/v1/libthread.so)
==26017==    by 0x5EAF000: kepsocket::IConnection::IConnection(kepsocket::CBaseSocket&, kepsocket::data_monitor_instance_t)
==26017==    by 0xCDFE4D6: kepua::CConnection::CConnection(std::shared_ptr<kepua::ITransport>, kepsocket::CBaseSocket&) (in
==26017==    by 0xCE3CD12: kepua::tcp::CSecureChannelLimiter::AddNewConnection(kepsocket::CBaseSocket&, kepua::tcp::transpc
==26017==    by 0xCE3D62B: kepua::tcp::CEndpoint::OnClientConnect(kepsocket::CBaseSocket&) (in /opt/tkedge/v1/libua.so)
==26017==    by 0x5EB24EB: kepsocket::CSocketServer::_Accept(unsigned int) (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0x5EB2591: kepsocket::CSocketServer::_ConnectionMgr::OnTimeout() (in /opt/tkedge/v1/libsocket.so)
==26017==    by 0x69DF074: kepthread::CKEPThread::_ThreadProc(void*) (in /opt/tkedge/v1/libthread.so)
==26017==    by 0x50476DA: start_thread (pthread_create.c:463)
==26017==    by 0x869788E: clone (clone.S:95)
```

Figure 7 Valgrind log showing the malloc, free and use after free.

Key take aways

- This deep dive into examining the security of the OPC protocol is vital because OPC isn't going away any time soon. OPC is often the communication hub of an OT network, centrally supporting communication between proprietary devices that otherwise could not exchange information. It's deeply embedded in many product configurations, and OPCcentered development figures to continue.
- Vendors are already connecting OPC, which extracts data from control systems and communicates that information to other systems on the shop floor, to the cloud. This introduces industrial IOT devices into the equation, both receiving and exchanging device and process information. Attack surfaces will expand and organizations must examine their respective implementations for weaknesses, and the community must support enhanced security and research into undiscovered vulnerabilities and protocol shortcomings

- At the core of these issues is secure code development, and static and dynamic code testing. Vendors must institute and adhere to a secure development standard; fuzz your own code, red-team, and pen-test your software for vulnerabilities. Implement processes to address security vulnerabilities in a timely manner, and when possible, share that work with the ICS and OT community
- Every piece of software will have vulnerabilities; uncovering them internally or working with security companies and independent researchers to find bugs is crucial to a secure ICS ecosystem. Cooperation with CERTs is also vital because these entities act as a central clearinghouse for coordinated disclosures, private secure communications with vendors, and prompt sharing of vulnerability information.

Conclusion

An attacker can send special messages that will cause an application to crash. This is usually the result of a known problem in a stack or application. These system bugs can allow a *client* to issue a command that would cause the *Server* to crash, as an alternate it might be a *Server* that can respond to a legitimate message with a response that would cause the *Client* to crash. The attacker could also be a *Publisher* that issues a *Message* that would cause *Subscribers* to crash

Reference

[1] https://opcfoundation.org/

[2] https://en.wikipedia.org/wiki/OPC_Unified_Architecture#cite_note-3

[3] https://attack.mitre.org/

[4] https://condor.depaul.edu/ichu/ds420/lecture/1030/public_key_encryp.htm

[5] https://www.claroty.com/ wp-content/uploads/2021/02/FINAL_Claroty_OPC_Research_Paper.pdf

[6] https://profiles.opcfoundation.org/v104/reporting/

[7]https://us-cert.cisa.gov/ics/advisories/icsa-20-352-02

[8] https://www.researchgate.net/publication/336765682_Simulating_and_Detecting_Attacks_of_Untrusted_Clients_in_OPC_UA_Networks

[9]https://www.google.com/urlsa=t&source=web&rct=j&url=https://arxiv.org/pdf/2104.06051&ved=2ahUKEwjMuaGiqfHxAhVBt6QKHXyQCC4QFjALegQICxAC&usg=AOvVaw3D0iQWkFDUxkzLdTWXqYO9