EXPLORING THE OPC ATTACK SURFACE

The Claroty Research Team presents its examination of the security of different implementations and usages of the OPC protocol stack, which standardizes communication between proprietary protocols on OT networks.

By Uri Katz and Sharon Brizinov

CLAROTY

CONTENTS

EXECUTIVE SUMMARY

- OPC is a network protocol stack that standardizes the way proprietary industrial devices exchange information about processes.
- OPC is often the hub of communication on an OT network, and this paper provides a deep analysis of the different protocol specifications, how they work, difficulties in implementing them, security challenges, and recommendations for defenders.
- During its examination of OPC, Claroty uncovered several critical vulnerabilities in vendor implementations of the OPC protocol that could expose organizations to remote code execution, denial-of-service conditions on ICS devices, and information leaks.
- Claroty privately disclosed its findings to industrial automation vendors Softing Inc., Kepware PTC, and Matrikon Honeywell. All three have provided fixes and users are urged to update to the latest versions of each affected product.
- The Industrial Control System Cyber Emergency Response Team (ICS-CERT) has also published advisories for each set of vulnerabilities, warning users of the affected products about the risks. Update and mitigation information is also available from ICS-CERT in the following advisories: Softing Inc., Kepware PTC, and Matrikon Honeywell.
- The affected protocol libraries, gateways, and server software are all embedded in vendor offerings as third-party components, further exposing the potential risk posed by these vulnerable code bases. Companies must determine whether their products are based on these vulnerable OPC implementations and update accordingly.
- As more researchers from security companies such as Claroty, as well as independent researchers, increase their scrutiny on ICS and SCADA devices, additional vulnerabilities across the domain are expected to bubble to the surface.
- Other innovations such as connecting OT networks to the cloud will expand the OPC attack surface, and force organizations to be vigilant about their vendor implementations and secure coding practices.

INTRODUCTION

Claroty researchers in 2020 conducted an extensive analysis of the OPC network protocol prevalent in OT networks worldwide. During that research, Claroty found and privately disclosed critical vulnerabilities in OPC implementations from a number of leading vendors that have built their respective products on top of the protocol stack. The affected vendors sell these products to companies operating in many industries within the ICS domain.

OPC (Open Platform Communications) is a common network protocol used in products to monitor and control systems from different vendors whose proprietary communication protocols are incompatible. Due to its massive popularity, the OPC protocol has attracted a lot of attention from researchers and attackers. Many researchers are fuzzing the OPC protocol and producing beneficial results. For example, Kaspersky Lab published a detailed report in 2018 claiming to have identified 17 security issues in some well-known OPC UA implementations.

The vulnerabilities discovered by Claroty could be exploited to cause a denial-of-service condition on devices operating on industrial networks, as well as information leaks, and remote code execution. Our research identified weak spots in different OPC specification implementations within different components of the OPC architecture. These components include the OPC server, OPC gateway, and a third-party library implementation of the OPC protocol stack.

Specifically, in our research we focused on three products:

- Industrial Automation OPC library by Softing Inc.
- ThingWorx Kepware Edge and KEPServerEX OPC Servers by Kepware PTC
- MatrikonOPC Tunneller by Matrikon Honeywell

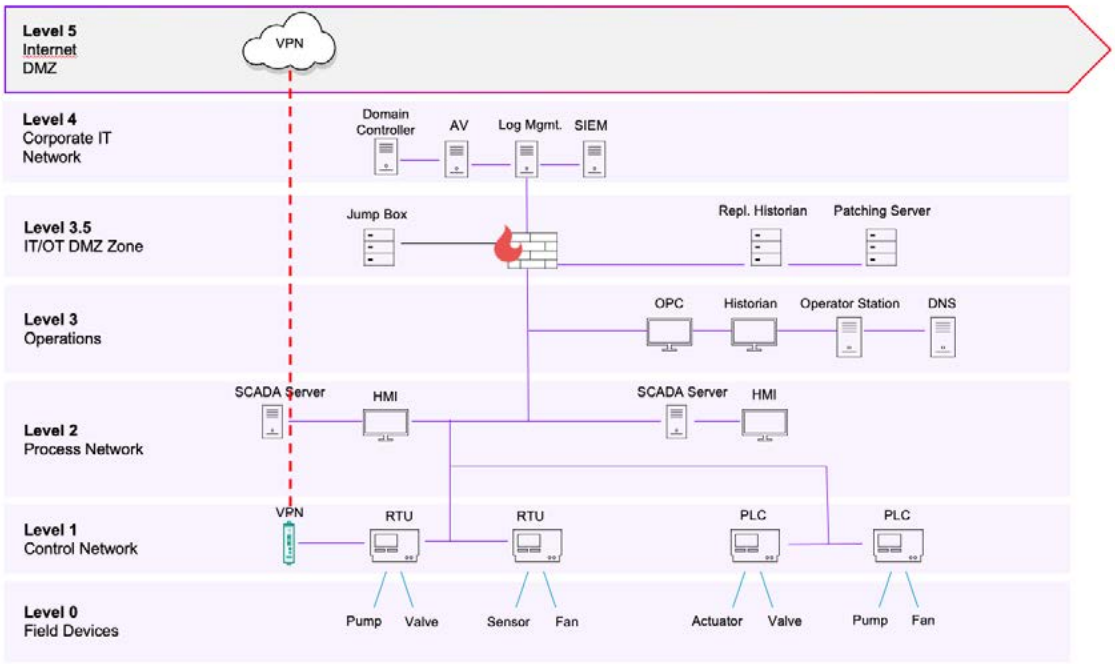
These three products are integrated into many other vendors' offerings as a third-party component. For example, Softing's OPC library is being used as a third-party OPC protocol stack by some vendors, and the KEPServerEX OPC Server is being used as an OEM shelf solution by other well-known vendors, including Rockwell Automation and GE, both of which have published advisories informing their users of these security issues. We believe these vulnerabilities affect multiple other products sold by vendors across all ICS vertical markets.

In this report, we will explain the OPC protocol in depth, its architecture, and common usage in order to gain a deeper understanding of the impact of these vulnerabilities. We will also describe the vulnerabilities we uncovered, and explain the potential threat posed by attackers who exploit these vulnerabilities to take over OPC servers and gateways, and potentially harm manufacturing facilities and production lines.

OPC EXPLAINED

WHAT IS OPC?

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.

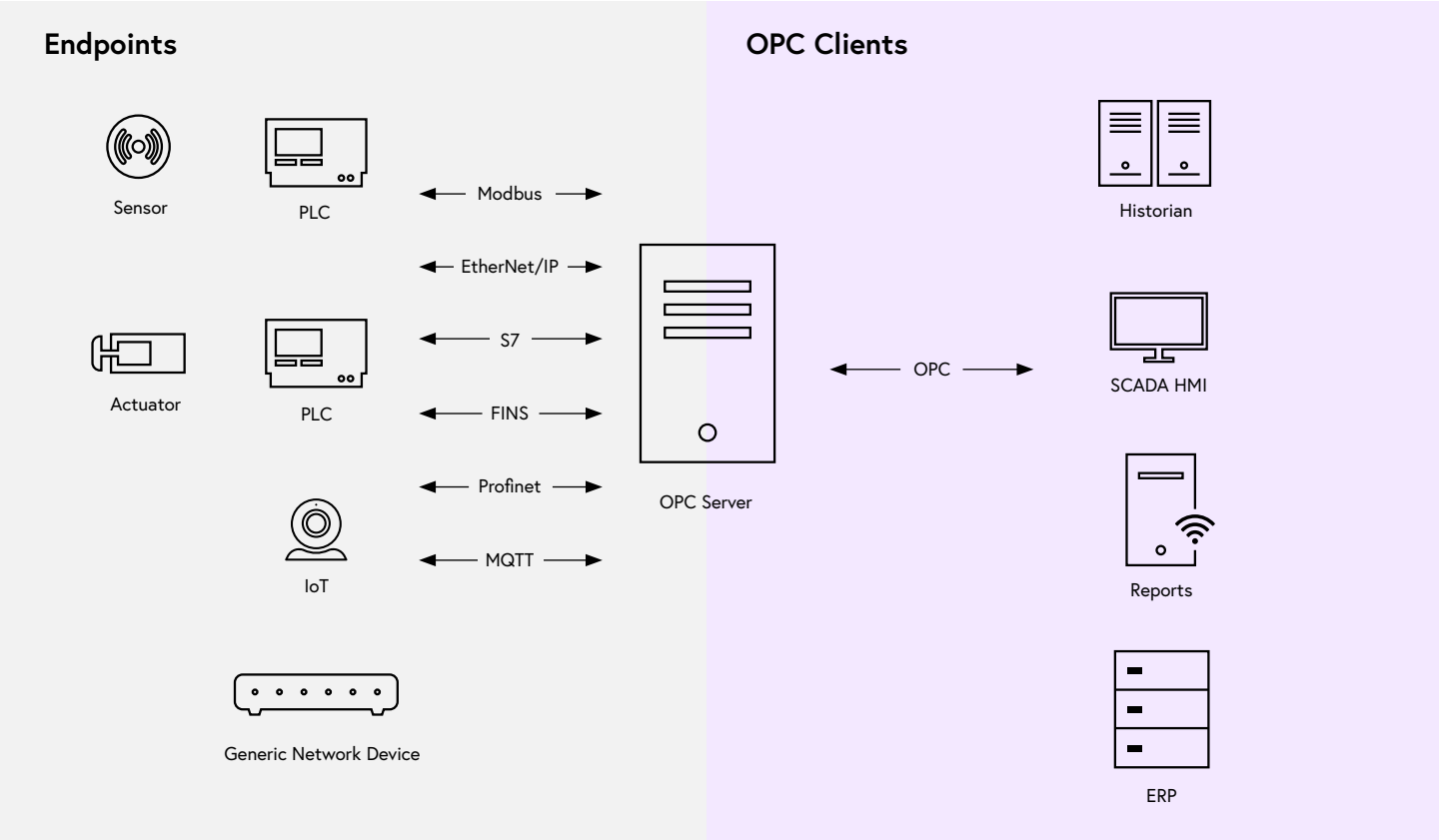


Purdue Model showing the various components of a typical OT network.

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.



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

HISTORY AND PROTOCOL EVOLUTION

The first generation of the OPC standard and specification was developed 25 years ago. It was created to abstract PLC-specific protocols such as Modbus and Profibus, for example, into a standardized interface. This would allow HMIs and other SCADA systems to interface with "a middle man" that would convert OPC read/write requests for proprietary devices. This gave birth to a segment of the ICS and SCADA markets filled with products that seamlessly interact through implementations of the OPC standard.

The original protocol, OPC Classic, enabled devices built on different specifications to exchange process data, alarms, and historical data. OPC Classic specifications consisted of:

- OPC Data Access (OPC DA)
- OPC Alarms & Events (OPC AE)
- OPC Historical Data Access (OPC HDA)
- OPC XML Data Access (OPC XML DA)
- OPC Data eXchange (OPC DX)

Each had its own read/write and other commands that applied to one protocol at a time, regardless of whether an OPC server supported multiple protocols. Data Access (DA) is the oldest and most common; it extracts data from control systems and communicates with other systems on a shop floor.

At its core, OPC Classic is built on Windows, using COM/DCOM (Distributed Component Object Model) for communication among software components in a distributed client-server network. Initially, OPC was Windows-only (its acronym comes from OLE, or object linking and embedding, for Process Control). OPC Classic is widely adopted in many sectors such as manufacturing, building automation, oil and gas, renewable energy, and utilities, among others.

As manufacturing and technology evolved toward service oriented architectures (SOA), data modeling and security challenges bubbled to the surface. This prompted the OPC Foundation to develop OPC UA (Unified Architecture) to address these issues and at the same time provide a feature-rich technology and open-platform architecture that was future-proof, scalable and extensible.

OPC FLAVORS

The original OPC Classic protocols had nothing in common. Each of the protocols—DA, AE, HDA, XML DA, and DX—have their own commands that apply only to the respective protocol, regardless if the OPC server supported several protocols. We will cover each in this section of the report.

DA (Data Access): The Data Access protocol extracts data from control systems and communicates that information to other systems on the shop floor. It represents actual device values such as timestamps, temperatures, and more.

AE (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.

HDA (Historical Data Access): HDA 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 (Unified Architecture): OPC UA, unlike OPC Classic, does not rely on Microsoft OLE or COM/DCOM for communication; therefore, it's cross-platform and can be used on various operating systems including Mac, Linux, or Windows systems. Also, structures or models may be used with UA, meaning that data tags or points can be grouped and given context, simplifying governance and maintenance. Models may be identified in runtime, making it possible for clients to explore possible connections by making a server request.

OPC UA supports numerous custom and proprietary transport capabilities, and must do so in order to be a scalable solution. This architecture can be found in everything from shop floor sensors to Windows servers.

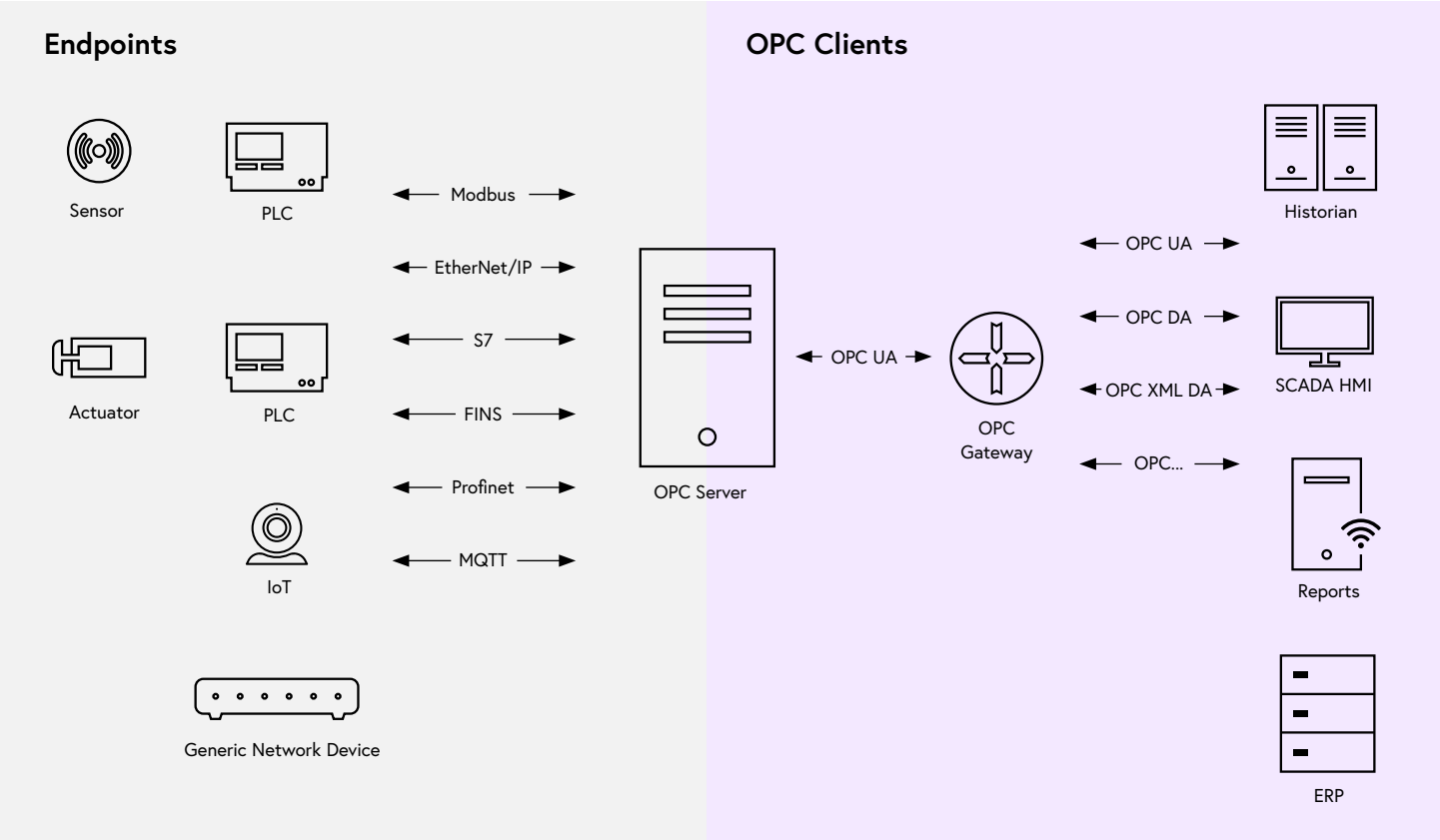
The OPC UA specification defines several transports that clients are required to support. This includes SOAP over HTTP/HTTPS or raw OPC UA over TCP transport. The latter is designed for server devices lacking the resources to implement XML encoding and HTTP/SOAP type transports. UA TCP uses binary encoding and a simple message structure that can be implemented on low-end servers.

OPC UA transports are powerful mechanisms that allow standardized messages to be sent between the factory floor, crossing the OT spectrum to communicate with IoT/IT devices. This is accomplished through OPC UA requests to read/write attributes using standard web services languages and protocols. For example, UA requests and responses can be encoded as XML inside SOAP requests and transferred to IT applications that are interoperable with these mechanisms.

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.

Our network diagram now looks like this:



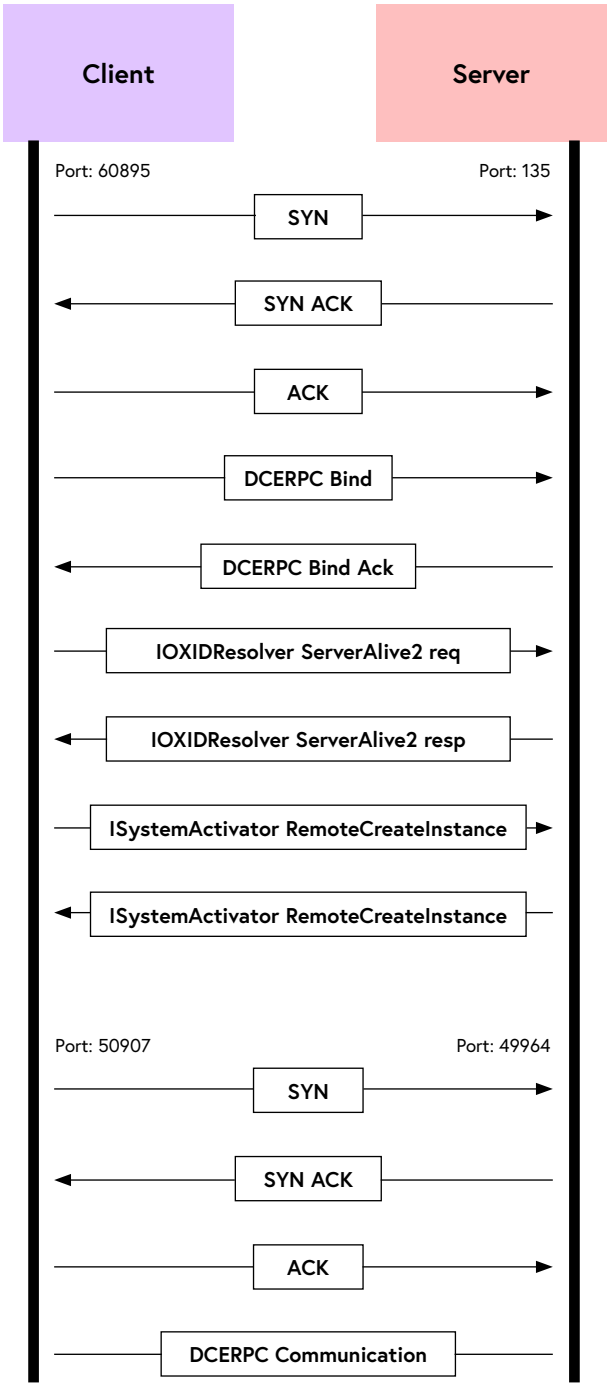
OPC gateways convert different specifications to the OPC protocol.

PROTOCOLS IN-DEPTH

OPC DA

As mentioned earlier, 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.



DCERPC communication initialization process.

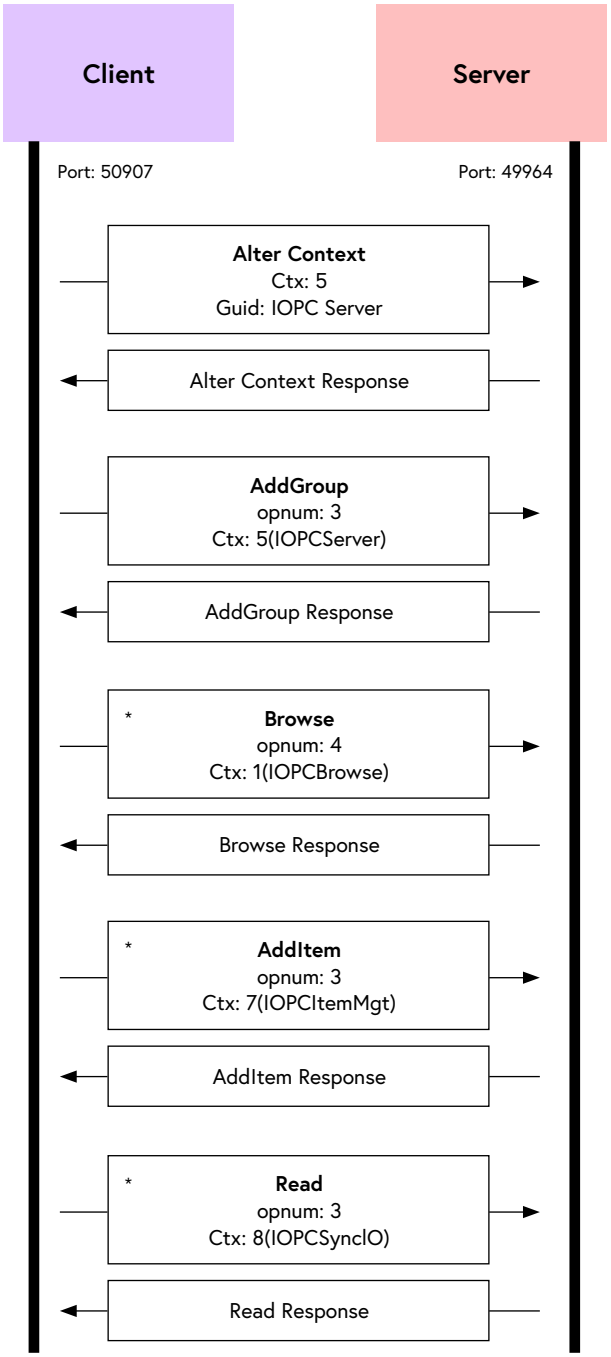
The OPC protocol, like all COM-based protocols, has pre-built interfaces which describe the OPC functionality that can be called remotely. After the initial connection is made, the client maps these interfaces, which are identified by their UUID, to a Context ID using the AlterContext method. After the 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.

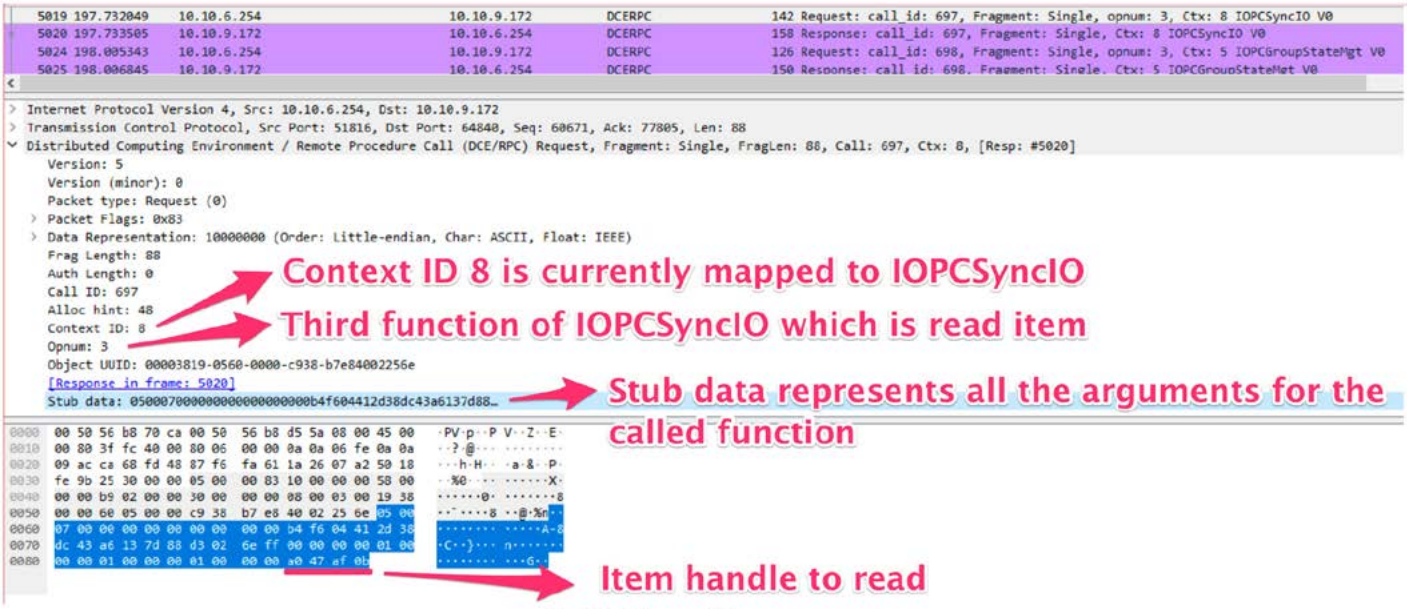
Given all this information, here is a typical OPC DA packet sequence to read an item from the OPC server:



OPC DA example communication flow.

Please note that Alter Context occurs before accessing a new interface, denoted with a (*).

And here is a an example of OPC read item request as we captured in wireshark:



OPC DA read item request.

OPC UA

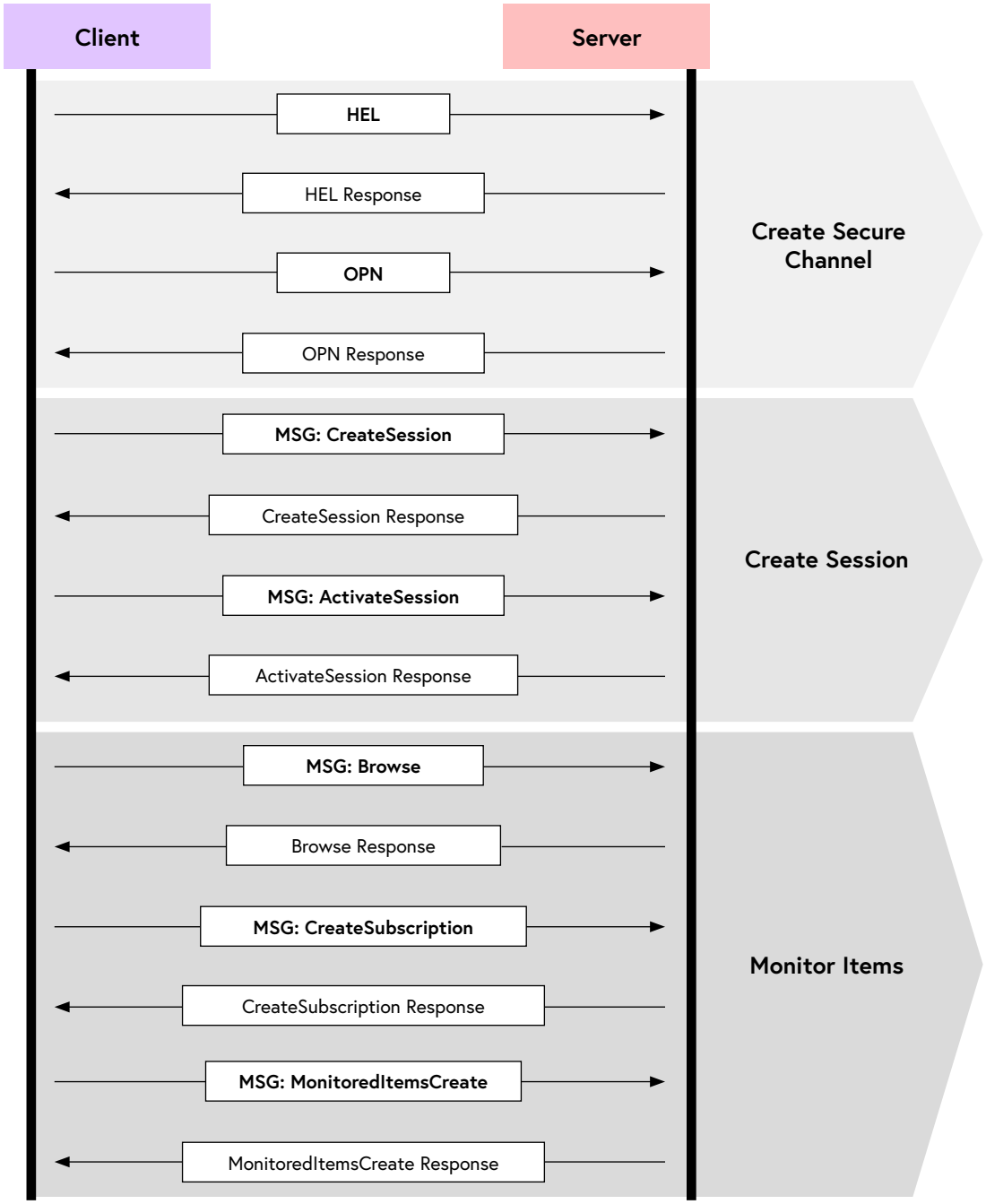
Every OPC UA message starts with a three byte ASCII code that makes the message type. The four main message types are:

- 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.

Here is a typical example of an OPC UA packets sequence flow to monitor items' values:



OPC UA sample communication flow.

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.

CLAROTY'S RESEARCH

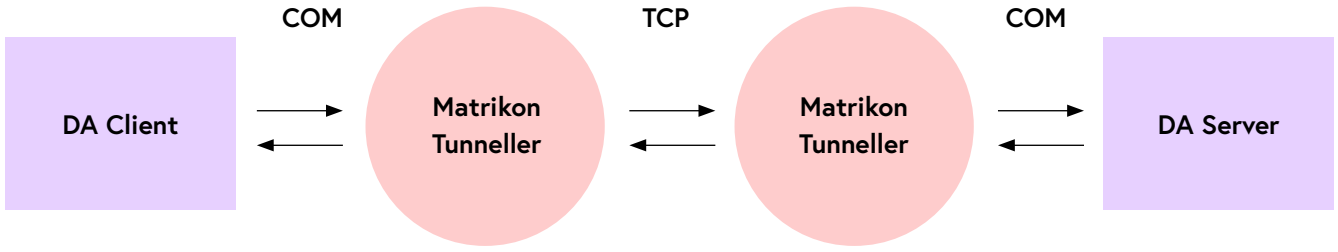
OPC DA VULNERABILITIES

OPC DA is built with COM/DCOM as its communication method between clients and servers. The DCOM marshalling and unmarshalling of data ensures the server and client get the data in the correct structure. The parsing of malformed packets, for example when the reported number of items doesn't match the actual number of items in the packet, fails at the unmarshaling level and does not reach the OPC server logic. Implementing proprietary protocols that skip the COM/DCOM handling exposes the server directly to user-provided data. We decided to focus on such cases to understand what are the common mistakes when implementing complex OPC DA data structures parsing.

MATRIKON'S OPC DA TUNNELLER

Matrikon OPC Tunneller is a DCOM-less solution for OPC servers that also enables you to combine OPC Classic and UA servers by converting OPC Classic traffic to its proprietary protocol.

Matrikon Tunneller has a few configuration options, but we are going to focus on the Classic Client to Classic Server configuration where client and server are configured to talk to the local tunneller via COM. The client DA data will be translated to the Matrikon protocol and sent over TCP to the server tunneller. The server tunneller will parse the Matrikon protocol and convert it back to OPC DA.



Claroty found multiple heap overflow vulnerabilities in different Matrikon OPC Tunneller components that could allow for remote code execution on affected machines. In addition, other conditions could be exploited that would result in denial-of-service attacks on devices because of excessive resource consumption or improper checks.

CVE-2020-27297

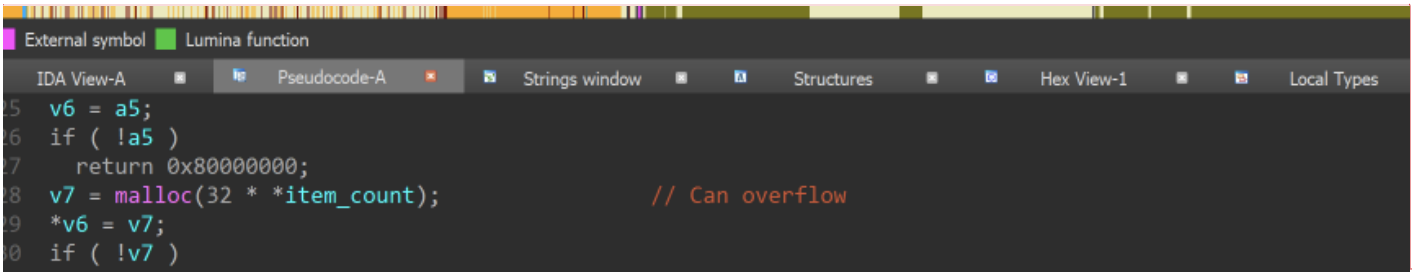
MATRIKON OPC TUNNELLER: HEAP OVERFLOW DUE TO INTEGER OVERFLOW

In this case, we decided to focus on the decoding/encoding of the proprietary protocol of Matrikon OPC Tunneller. The Matrikon protocol is similar to the OPC DA protocol in many ways because eventually it tries to achieve the same goal: transfer OPC objects over the network. Most action commands, including read_items, write_items, add_items, receive an array of structured items to parse. The array can contain items of different types.

Many Matrikon OPC functions follow the same handling pattern:

1) First, they decode the item_count value from the packet.
2) Next, they allocate memory by performing malloc in the size of the product of item_count value * item_size.
3) Lastly, they decode the items and store the objects in the allocated buffer.

We found an issue with the allocation mechanism of the decoded OPC items. Here is a pseudocode of the memory allocation based on item count:



Allocation of memory based on item count in Matrikon AddItems function.

We can see that the field that represents the number of items from a given packet is multiplied by the item struct size, for example in this case, it's 32 bytes. The item_count field is a 32-bit unsigned integer which is controlled by the user.

Since there is no verification or sanitation regarding the reported item_count from the packet, in 32-bit architecture machines, this malloc is susceptible to an integer overflow vulnerability in case of a huge item_count. The result is a memory allocation that will be much smaller than needed. For example, if we send a packet with item_count 0x80000001 and the item size is 32 bytes, the result would be a malloc of just 32 bytes instead of 0x80000001*32 = 0x1000000020 bytes because of the 32 bit overflow.

After mallocing the buffer, the items are parsed in a while loop using the original item_count as the exit condition, which means that the loop will run 0x80000001 times unpacking items into a buffer of 32 bytes. Hence, a heap buffer overflow occurs with data at our control.

CVE-2020-27299

MATRIKON OPC TUNNELLER: INFORMATION LEAK DUE TO AN OOB READ

Claroty also found heap out-of-bounds (OOB) vulnerabilities in the Matrikon OPC Tunneller which an attacker could exploit to force a memory leak. The leak could enable an advanced attacker to carry out other exploits on the network.

Keeping our focus on the translation from the TCP protocol to the program objects, we looked at a function that seemed to decode item types, meaning it takes a binary buffer from the packet and decodes it into a specified type. We named this function DecodeType. This function plays a crucial role in the decoding of the OPC-like protocol by Matrikon due to the fact that every protocol command reaches its parsing function and there the DecodeType is called with predetermined types. For example, read and decode the first two bytes to a short integer, the next four bytes are of type integer, etc.

Going through the protocol commands, we found what looks like an old implementation of the DecodeType function. The old function is mostly the same as the new one, but is missing some of the checks and verification that the main function has. One such missing check determines whether the end of the buffer has been reached, and instead it completely trusts the user-controlled data.

We were able to exploit this missing check in two ways. The first method we used was to send a big item_count which caused the while-loop to keep decoding arbitrary memory values into a buffer that later was sent to the server, thus leaking information from the Tunneller's heap memory to the DA server.

The second approach we used was to send a packet with a special item type with less-than-needed payload data. This packet forced the old DecodeType function to read data from the heap while passing the end of our packet heap chunk. In other words, the function reached an out-of-bounds read condition and again it kept on decoding arbitrary memory values into a buffer that later was sent to the server, thus leaking information from the Tunneller's heap memory to the DA server.

OPC DA XML VULNERABILITIES

OPC DA XML is a subvariant of OPC which is transferred over HTTP SOAP. As the name implies, it is built on XML entities instead of a binary format. Like the Matrikon tunneller, OPC DA XML provides a different transport layer than the classic COM/DCOM. This increases the server's attack surface. The server needs to both self implement a transport layer, and verify and parse the data directly from the user.

SOFTING'S OPC DA XML LIBRARY

Softing has been developing for more than 30 years products that support industrial communication using fieldbuses, industrial Ethernet protocols, and data exchange. The company offers development toolkits for OPC Classic and OPC UA, and end-user products.

Claroty discovered two vulnerabilities in the handling of OPC DA XML. One vulnerability targets the transport layer, the HTTP SOAP server, and the other targets the XML data. Both vulnerabilities lead to denial of service (DoS) and are trivial to exploit.

CVE-2020-14522

SOFTING OPC DA XML: DENIAL-OF-SERVICE DUE TO UNREACHABLE EXIT CONDITION

Just like the regular OPC DA protocol, a handle is used to identify items. The ClientItemHandle parameter is expected as a hexadecimal value formatted as _x[value] (e.g. _xff). In case a _x value is found, the function checks if the next four bytes are hexadecimal values and if so they will be parsed and the pointer to the buffer will be increased to the next data to be parsed. However, if the read tag is not a valid hexadecimal value, the code will not increase the pointer and get stuck in the loop forever. This means that this loop will run indefinitely and triggering it from multiple threads will cause high memory consumption and a denial-of-service condition.

```
POST /SDG HTTP/1.1
User-Agent: Softing OPC Toolkit
Host: 1.2.3.4:8081
Date: Mon, 23 Dec 2019 08:40:08 GMT
SOAPAction: "http://opcfoundation.org/webservices/XMLDA/1.0/
Subscribe"
Connection: close

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope
" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="
http://www.w3.org/2001/XMLSchema" xmlns="http://opcfoundation.org/
webservices/XMLDA/1.0/">
<soap:Body>
<Subscribe ReturnValuesOnReply ="true" SubscriptionPingRate="5000">
    <Options ReturnItemTime="true"/>
    <ItemList RequestedSamplingRate="1000">
        <Items ItemPath="" ItemName="Test" ClientItemHandle="
        _x_x_x_x_"/>
    </ItemList>
</Subscribe>
</soap:Body></soap:Envelope>
```

SOAP packet that triggers the denial-of-service.

CVE-2020-14524

SOFTING OPC DA XML: DENIAL OF SERVICE DUE TO HEAP EXHAUSTION

Softing's web server receives SOAP requests and parses them as OPC DA XML over SOAP. However, the server does not limit the SOAP headers' length nor sanitize its values. This means that if we send a very long header (SOAPAction, for example), the server will endlessly keep on allocating memory without any limitation.

At some point, the memory allocation will fail due to resource consumption of the heap memory. However, the web server does not check the return code of the memory allocation and tries to copy our data anyway to the returned pointer. But since the returned pointer is NULL our data is copied to uninitialized memory, eventually causing an access violation exception and a crash of the server.

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-of-service 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

Claroty 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:

For example, the HEL message is constructed as follows:

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);
    }
  }
}
```

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

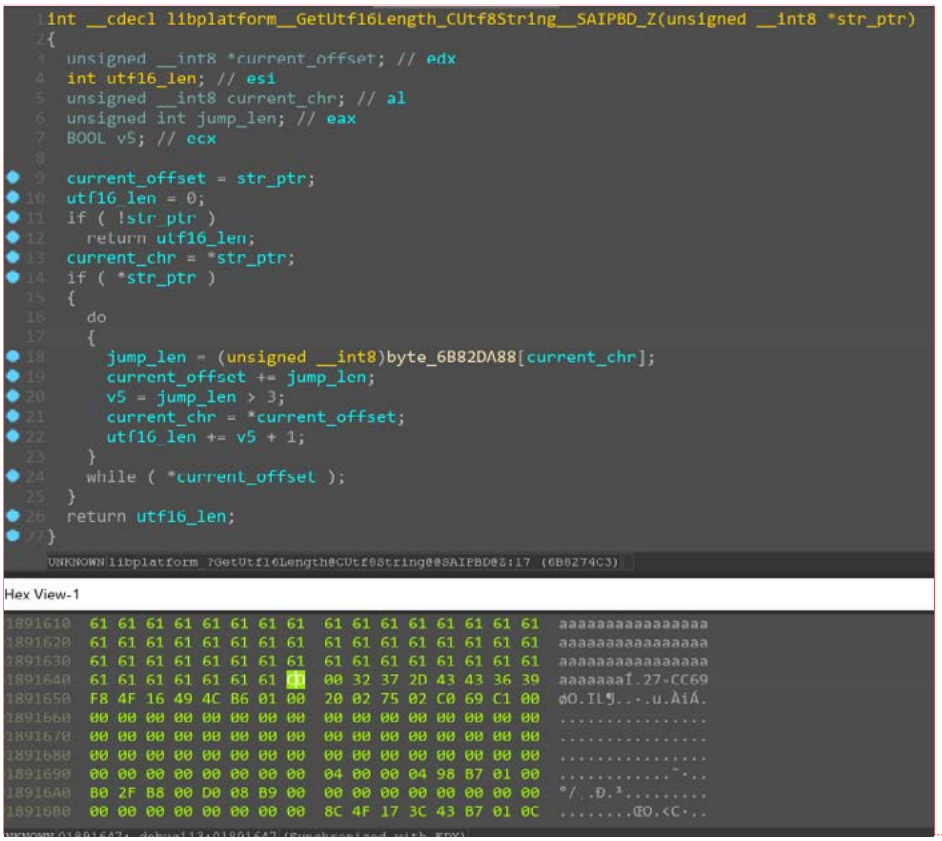
KEPWARE THINGWORX EDGE SERVER: INFORMATION LEAK DUE TO A HEAP OOB READ

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.



CUtf8String::GetUtf16Length function and memory values while debugging the function.

CVE-2020-27267

KEPWARE KEPSERVEREX EDGE SERVER: DENIAL OF SERVICE 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.

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

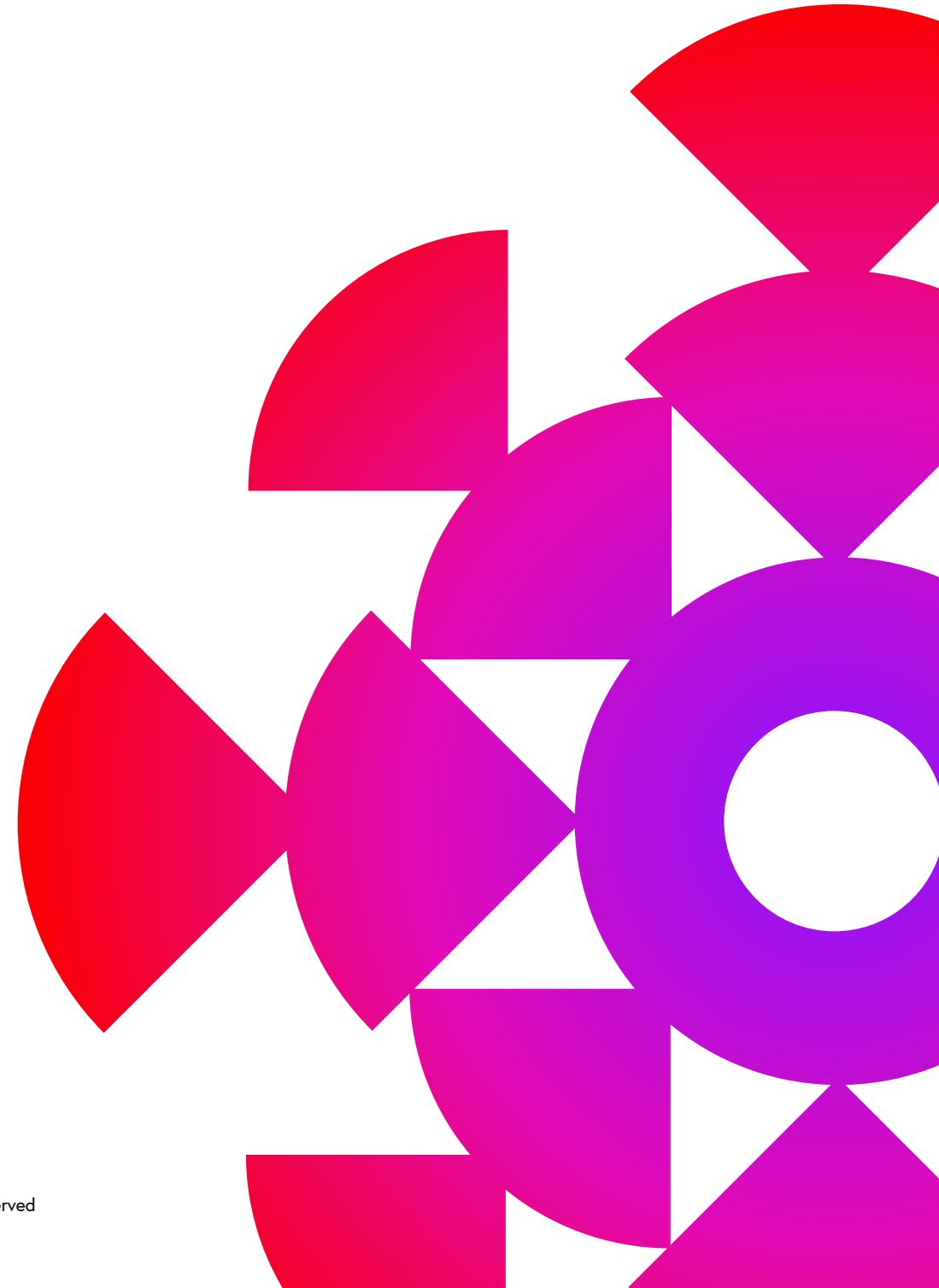
KEY TAKEAWAYS

- 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 OPC-centered 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.



CLAROTY