# OPC Exposed – Part I

Lluis Mora
Neutralbit
World Trade Center
Edifici Sud, Planta 2
08039 Barcelona, Spain
E-mail: llmora@neutralbit.com

**Abstract:** OPC (OLE for Process Control) is a mechanism for interconnecting process control applications running on Microsoft platforms. It is layered on top of DCOM which in itself presents a wide range of security considerations. OPC is defined in a series of specifications, although the best known and more widely implemented is the Data Access (DA) specification.

We present a security analysis of the DA specification, identifying theoretical weaknesses that implementers should take into consideration when developing OPC clients and servers. To validate our findings a vulnerability group test has been conducted against several OPC servers, discovering a number of previous unknown vulnerabilities.

Although MSRPC services have been widely tested for security vulnerabilities, the tests have centered around the transport layer and not on the application layer that DCOM implements. OPC testing revolves around the study of the protocol and the identification of weaknesses, identifying various new vulnerabilities in the process. The main innovation in this paper is in analyzing DCOM from the interface point of view and automating discovery and testing of OPC servers.

**Keywords:** OPC, OPC Security Vulnerability Testing, SCADA

## 1 Introduction

OPC (OLE for Process Control) is a set of vendor-neutral specifications created by the OPC Foundation[1] to facilitate the interoperation of process control products. Of the various specifications produced by the Foundation, the "OPC Data Access" specification is the most widely known and implemented; this paper focuses on the "OPC Data Access 2.0" specification which will be referred to as "OPC" throughout the document.

The Data Access specification defines a way to access heterogeneous devices in a control network through a common set of interfaces; this specification is proprietary to the OPC Foundation which makes it available to its members so that they can implement OPC-compliant client and servers. The Foundation also makes available a

[1] http://www.opcfoundation.org

redistributable package which includes the IDL description of interfaces, specifying the interfaces and methods that each client and server has to implement.

The specification defines what functionality the clients and servers have to provide, but leaves up to the particular vendors the actual implementation of the interfaces. The present document is a study of the potential weaknesses an OPC server implementation may present, based on the study of the interfaces it needs to implement, previous experiences and several server implementation reviews.

Throughout the paper you will find that we describe *potential* vulnerabilities - after all, OPC is just the skeleton of what needs to be implemented and does not in itself contain any functionality that might lead to vulnerabilities. The weaknesses reside in the way some implementations decide to provide the functionality; it is with implementers in mind that the paper is structured as a series of security considerations to take into account when developing OPC servers.

## 2 Technology and Platforms

To connect clients and servers, OPC uses DCOM (a technology for distributed inter-process communications) which is in turn layered on top of MSRPC. Both DCOM and MSRPC are Microsoft proprietary protocols so most of the developed servers run on the Windows platform, although solutions exist to implement OPC on other platforms[2].

Servers which implement the OPC DA v2.0 specification need to expose the following COM objects and interfaces:

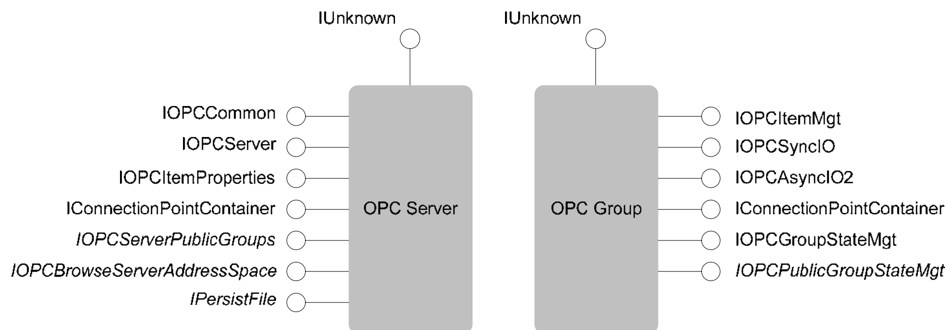


*Figure 1 – OPC objects and interfaces*

The focus of this paper is on the application layer of OPC; vulnerabilities of DCOM or MSRPC have been widely studied before and fall outside the scope of this document. OPC authentication is provided at the DCOM layer, in the following sections it is assumed the user has permissions to access the applications.

[2] See the "DCOM on Non-Microsoft Platforms" article on the OPC Foundation website

## 2.1 Server Fingerprinting

Identification of a remote OPC server through the network is built into the protocol: when establishing a connection the client needs to know the remote CLSID, a unique identifier for a specific server version.

The *OpcEnum* application, developed by the OPC Foundation and distributed with most servers, allows the remote enumeration of OPC servers running on a particular host. This application implements the `IOPCServerList` interface, whose `GetClassDetails` method returns for each OPC server its CLSID, along with the associated program id and a human readable description. Access controls for *OpcEnum* are usually more relaxed than for OPC servers, as server enumeration is required for the dynamic discovery of servers and interfaces.

After a client successfully connects to the server, it can obtain additional information by calling the `GetStatus` method of the `IOPCServer` interface. This method returns the following server data:

- Vendor information
- Version and build number
- Other information, such as current time, start time, server status, etc.

As it is part of the protocol, fingerprinting of OPC servers is trivial – down to a level (software, version and build) which leaves no margin for error.

```
C:\>opctest 10.0.1.23 list
[*] <Init> OPCTest v1.0 - NeutralBit (c) 2006

[*] <Init> Server #0: Matrikon.OPC.Simulation.1 (Matrikon OPC Server for Simulation and Testing) {F8582CF2-88FB-
[*] <Init> Version: 1.1 (build 307)
[*] <Init> Vendor: Matrikon Consulting Inc (780) 448-1010 http://www.matrikon.com
```

*Figure 2 – Remote server fingerprinting*

## 2.2 Server Handles

Once a connection is established the communication between client and server flows through the sequential remote invocation of server methods; the server does not keep track of "states" the client is in, so there is a need for a client to obtain unique identifiers to items and groups so that it can refer to them in further communications with the server. The handles speed up transactions as they can be used in place of string identifiers, which are expensive to parse and map to a particular group or item.

An OPC handle is an arbitrary, unique, 32-bit identifier that the server associates with particular groups and items. A client receives these identifiers after adding a new item or group (which is initially referred to using its full name) and is supposed to use the handle to further refer to the newly created element; for instance to write to a particular item the client needs to provide the item handle.

Servers usually store items and groups for a particular client as structures which are accessed using its memory address, a 32-bit value in the most common OPC platforms, so it is too tempting not to use the memory address of an item or group structure as the OPC handle. This approach speeds up access to the structure and avoids the maintenance of a handle-structure mapping table. In fact designers seem to have been indeed tempted, as most of the server implementations we reviewed used this technique.

The major drawback to this approach is that server handles are used as input parameters in client calls and an attacker can easily modify them. To protect the server address space integrity the server needs to verify the address location and type of client-provided server handles in each request, otherwise bad things can happen.

### 2.2.1 DoS Through De-allocation of Invalid Addresses

Calls to element deletion functions usually result at the server side with dynamically allocated memory being de-allocated using `free()` or a similar call. If the server does not validate server handles passed to `RemoveGroup`, `RemovePublicGroup` or `RemoveItems` it might end up trying to free non-reachable addresses, causing the server to crash.

### 2.2.2 Execution Through Dynamic Memory Corruption

Building on the previous example, if the server is does not validate de-allocating arbitrary memory, chances are it can be used to modify the internal data structures of the dynamic memory allocator, allowing arbitrary writes that can be leveraged to run attacker-provided code on the server[3].

### 2.2.3 Execution Through Arbitrary Memory Writes

Element removal functions are not the only functions that accept server handles as parameters. Specifically, the item write function takes two parameters: an item handle and a value to write to it. If the server maps handles to memory addresses and fails to validate a client-provided handle, the IO interfaces `Write` function allows an attacker to write any value to any memory address, a primitive which can be easily exploited to run arbitrary code on the server (e.g. through stack return addresses or SEH overwriting).

Here is a summary of the functions that use server handles provided by the client which, if the server implementation is faulty, can be used to read/write arbitrary memory addresses:

- `IOPCItemMgt::SetActiveState`
- `IOPCItemMgt::SetClientHandles`
- `IOPCItemMgt::SetDataTypes`
- `IOPCSyncIO::Read`
- `IOPCSyncIO::Write`

---

[3] http://www.phrack.org/archives/57/p57-0x09

- `IOPCAsyncIO::Read`
- `IOPCAsyncIO::Write`
- `IOPCAsyncIO2::Read`
- `IOPCAsyncIO2::Write`

## 2.3 Classic Overflows

OPC servers have to implement the body of the functions in the various interfaces they need to support. These methods have to handle strings, arrays and other types prone to overflows, something which all developers might not be familiar with, so checking implemented methods which accept these types might uncover apparently hidden vulnerabilities.

We have restricted the study to strings (the cornerstone of overflows) and integers (to find possible array index overflows) although a variety of other types might be as well considered. To automate the analysis a *fuzzer* was constructed to feed different combinations of values to the methods that accept strings or integers as parameters. The tests were then tailored to specific protocol requisites, to minimize invalid tests.

The handling of long strings does not seem to be a problem for the servers we reviewed, which also do not seem to be affected by "format string" vulnerabilities. On the other hand we encountered unexpected results with the handling of integers, specifically in the handling of error messages.

The `GetErrorMessage` method of the `OPCServer` interface accepts two integer parameters, one is the error number and the other allows the selection of the client "Locale ID".

The error number is a numeric value that identifies a particular error message, e.g. 1 for "Invalid parameter", 2 for "Server disconnected", 3 for "Unknown item", etc.

OPC supports the concept of "Locale ID", an identifier of the language and regional settings that should be used when processing requests from a client. It is most often used to return error messages in the appropriate language, but it also plays its part in the interpretation of client requests, such as in the format of strings or numbers. The "locale ID" meaning assigned by Microsoft, a current list of ID mappings to actual locales can be found at http://www.microsoft.com/globaldev/reference/lcid-all.mspx.

Our interest in this function is that, in its implementation, it is common to retrieve error messages from an array, indexed both by error number and by locale id, such as in the following code:

```
...
int errno, lcid;
char *message = errorTable[lcid][dwError];
...
```

If the implementation does not do proper bounds checking on the error message ID or the locale, then it might be possible for an attacker to access arbitrary memory contents in the server as well as cause the server to crash by accessing unreachable memory.

## 2.4 Configuration Persistence

The OPC specification defines `IPersistFile`, an optional interface which can be implemented by those servers willing to give the remote client control over the server configuration.

This interface is exposed so that clients can manage a server configuration exclusively through OPC, avoiding the need to use external protocols and interfaces, and it introduces, amongst others, methods to load and save the server configuration files. The configuration stored in these files is completely server dependent. They are supposed to handle the backend configuration and do not store client group/item configurations or other temporary data.

From the various methods defined in the interface, `Load` and `Save` are possibly the most interesting for an attacker. Both methods accept a filename as their first parameter, something an attacker can use to make the server load the configuration from any file or to write the current configuration to disk. Considering that the server usually uses the Win32 API (e.g. `CreateFile`) to access files, the network redirector can be used to read/write from network locations that the attacker controls(e.g.`\\MACHINE\file.cfg`).

A number of methods can be leveraged by an attacker to conduct various attacks.

### 2.4.1 DoS by Loading an Invalid Configuration File

If the server discards the current configuration in favor of the one contained in a file provided by the attacker, and this file ends up being invalid, the server has to be able to fall back into the previous configuration. Failing to do so might leave the server with no configuration, unable to answer legitimate clients requests. Note that the attacker does not even need to provide a custom file, making the server load any well-known invalid configuration file (e.g. `c:\boot.ini`) might as well work.

### 2.4.2 Execution by Loading a Specially Crafted Configuration File

Configuration management is probably one of the more complex parts of software development: parsing a custom configuration format, allowing for hand-edited changes, being resilient to invalid entries in the file while continuing to properly load other sections, etc.

Behind the task of loading the configuration is always a method that reads and parses the file, differing in complexity based on the configuration format, although it is always tricky to implement properly. If the attacker is able to uncover a security problem in the parsing code, the `Load` method can then be used to load a specially crafted file which exploits the vulnerability. This file can be hosted at a machine controlled by the attacker and simply loaded via a network redirector filename.

### 2.4.3 Download Server Configuration to a Remote Location

The `Write` method also accepts a network path filename, so the current configuration can be downloaded onto a remote location. The attacker then has access to the server configuration, including any confidential information such as user credentials or network information that might be stored there. This is often the case in OPC servers that act as gateways or concentrators for other protocols. Furthermore, if the server has previously loaded an invalid file and keeps a reference to it, it might be possible to copy the file to the remote location by saving it there, e.g.:

1. The attacker calls `Load("server_confidential_file.dat")`
2. The server notes down its current configuration file is "`server_confidential_file.dat`"
3. The server fails to load the configuration and returns an error
4. The attacker calls `Save("\\MACHINE\destination.dat")`
5. The server copies its current configuration file, "`server_confidential_file.dat`" to the destination of the save operation

### 2.4.4 DoS by Overwriting a System File

The easiest and more trivial denial of service attack forces the server to overwrite a system file with the current configuration using the `Write` method. This could invalidate the overwritten file from properly loading in the future – e.g. make the server save the configuration file to the path of the executable file that provides the server functionalities; subsequently whenever the OPC server is invoked, it will not load and will therefore stop serving client requests.

As the file is written using the server current credentials, the attack can be extended to write files in other machines that happen to trust the OPC server credentials, possibly into machines the attacker has no direct access to.

Other methods from the `IPersistFile` interface include `GetCurFileName`, which returns the current path of the server configuration, allowing an attacker to gather information on the server environment, e.g. server drives and paths.

Experimenting with server implementations, it seems that only a small percentage of servers implement this optional interface, although it is good to know if a particular server does implement it to add extra security measures.

## 2.5 Type Conversions

Items have an associated data type which defines what type of value they hold (integer, string, etc). The opaque VARIANT type is used to simplify the number of methods each interface implements when data values are exchanged between clients and servers. The VARIANT type has a particular quality that allows it to hold various data types in itself (the VARIANT type is composed of a flag that indicates the data type it holds, along with the serialization of that value).

The use of VARIANT allows the specification to be simplified, providing just a single interface to read/write multiple types; there is no need to have separate operations to work on integers, operations to work on strings, etc.

Although an item has an associated date type, the client can use different VARIANT types in method calls that work with that item, as the server will, in most instances, try and convert from one type to another. In the event that types are not compatible, the server will signal so back to the client, which will need to retry the operation with the proper type.

The Windows API provides a complete set of functions to work with VARIANT types, including functions to convert between different data types (`VariantChangeType`). These functions abstract the application from any serialization and validation, providing a high-level interface to VARIANT types. This API is well known and tested, although some servers might choose not to use the standard API and implement custom VARIANT handling functions (for instance because only a small subset of data types are supported by the server, or because they need to keep a small memory footprint).

Creating a custom implementation of the VARIANT type API is a complex and prone to errors task: serialization / deserialization, type conversions, conversions that behave differently based on the current locale, etc. This complexity makes OPC type handling a likely spot where weaknesses could be introduced.

Particular interest needs to be paid to the methods in charge of deserializing a provided VARIANT, as an attacker might decide to provide an invalid serialization or one that does not match the indicated inner type (e.g. using a string serialization for an integer type). Also the more exotic types, such as VT_DISPATCH are rarely used, so their handling functions might not have been properly tested – at the very least conversions from such types to more standard types might prove tricky for the server.

Methods to look for in these cases are those that allow a client to specify a particular VARIANT type, namely the data writing methods:

- `IOPCSyncIO::Write`
- `IOPCAsyncIO::Write`
- `IOPCAsyncIO2::Write`

## 2.6 Resource Starvation

Availability is one of the main design principles in control system networks; data needs to be immediately accessible at all times. Attacks on availability are usually targeted at making the server crash, although introducing non-deterministic delays in responses is another equally successful approach to service denial. These delays can be caused by the starvation of server resources (usually network or CPU).

An attack on the CPU usage is usually easier and requires fewer resources on the attacker side; the attacker only needs to identify CPU intensive methods provided by the server, such as the linear walk of huge lists or the parsing of complex structures. OPC

servers offer various interfaces that, without the proper checks, can be used by an attacker to starve CPU resources:

- The `IOPCItemProperties` interface is not optimized to access a large amount of item information. Repeatedly calling its `QueryAvailableProperties` on different item IDs can increase CPU utilization.
- The optional `IOPCBrowseServerAddressSpace` interface method `BrowseOPCItemIDs` can select items based on a client-provided filter. Although each server chooses how to interpret this filter format, there is a reference implementation that makes the filter format similar to that of the SQL LIKE operator. An unusually complex filter such as '%%%%%%%%%%%%%' can make the filtering function take a significant performance hit.

Depending on the implementation of these functions the attacks can lead to increased memory consumption, resulting in out of memory errors that can ultimately crash the server. Otherwise the attacks will increase usage during a period of time, after which operations will be back to normality.

To prevent against these attacks clients can implement resource control limits for each client, along with timeout and client disconnection detection, so that only queries for legitimate clients will consume resources.

## 2.7 Blob Poisoning

Blobs are pieces of server-generated information that are sent along with item responses to the client. A client is expected to send them back in further requests to the same item so that the server can speed up its processing. In this respect blobs are similar to HTTP cookies, pieces of information that are reflected off the client just to maintain session state whose content the client does not need to understand.

Blobs are exclusively used to improve server performance, by containing information (which is specific to each server) that allows for faster item lookup and response. For instance, a server implementation might include as part of the blob a memory dump of the structure that holds item information, or information on when will the item be next updated, etc.

Blobs differ from server handles in that a blob can handle an arbitrary amount of information and that their usage by both servers and clients are optional.

A rogue client can modify the blob value before sending it back to the server, if the server does not check the integrity of the client response it can be misled to perform unexpected operations. If the blob holds any kind of structure, the server will need to parse any incoming blob – again adding complexity that might lead to security weaknesses.

Even if the structure is successfully parsed, its content can leave avenues open to attack. Take for instance a server blob that holds as part of its structure a memory pointer to an item value used in a Read / Write method. It can be leveraged to read and write to

arbitrary memory, much in the same way as happened with sever handles. In fact most of the attacks previously discussed on exploiting server handles are applicable to blobs.

Methods that directly or indirectly accept blobs provided by a client are related to item management:

- `IOPCItemMgt::AddItems`
- `IOPCItemMgt::ValidateItems`

Less than 5% of the servers we reviewed make an active use of blobs, so it does not seem to be a capability particularly popular with implementers.

### 2.8 Client callbacks

The OPC protocol defines the optional usage of client callbacks to improve performance and responsiveness; clients can register hooks for particular events (such as `OnDataChange` or `OnReadComplete`) and will receive a notification each time the server generates them, often as the result of an asynchronous read or write operation but also due to subscriptions to specific events.

Callback objects are provided by the client implementation of `IOPCDataCallback`, `IAdviseSink`, `IDataObject` or `IConnectionPoint`; whenever the server encounters an event that has a registered callback it will use DCOM to instantiate the interface on the client and subsequently remotely call the instance method, reversing the roles of client and server - with callbacks the DCOM is used in a two-way communication.

Although most of the risks associated with callbacks are client-side (and outside the scope of this document), the fact that the server uses DCOM to call the method on the client includes connection establishment and possibly authentication; depending on the server settings a client can force the remote server to authenticate before the callback, accessing user and server details (username, system name, domain…) and even authentication credentials.

Also if the server uses a specific worker thread to notify various clients of event notification, a non-responsive rogue client could block the operation of the thread, possibly leaving other clients expecting notifications that will never arrive.

## 3 Conclusions

Our review of 21 sample OPC servers to validate our findings is not very comforting; over 20% of the tested applications present at least one of the weaknesses we have highlighted.

These are the results of our OPC server group test:

| | |
|---|---|
| Server handle poisoning | 14.3% |
| Overflows | 9.5% |
| Configuration persistence | 14.3% |
| Type conversions | 4.8% |
| Resource starvation | 23.8% |
| Blob poisoning | 0,0% |

We believe these results are not acceptable, considering that these are applications with a critical job to do. We have purposefully avoided analyzing the security of DCOM or MSRPC, which fall beyond the control of the OPC server vendor, and have focused in weaknesses that have its only source in faulty implementations, so there is no one else to blame. Of particular interest are servers which, even though they detected memory access violations through exception handling, did nothing to stop the attacker from compromising the process address space.

Of course, we can not generalize – most of the servers stood unaffected after our test runs, but the fact that we were able to crash that many servers with an automated tool opens up the question of what could be achieved in a more targeted attack.

OPC server vendors need to go beyond the idea that DCOM security is handled by just setting DCOM permissions and take a more proactive approach towards security, embedding it in their development process and then verifying the security compliance of products and their resilience to attacks.

---

**About the Authors** – Lluis Mora is CTO at Neutralbit, an information security research and development company. He is in charge of the definition of new services and leads the Neutralbit Research & Development labs, which focus on the security assessments of information systems.

Before founding Neutralbit, Mr. Mora was CTO at Sentryware, an information security vendor. Previously he had founded S21SEC, a Spanish security services company, where he led the R&D department.

Mr. Mora has a BsC in Computer Science by the UOC from Barcelona and has published various articles on vulnerability research in information systems. He won the Openhack competition both in 1999 and 2000 and is member of the technical committee of AVDL and a member of OWASP.