Kaspersky ICS CERT

kaspersky

Practical example of fuzzing OPC UA applications

Pavel Cheremushkin

19.10.2020 Version 1.0

Contents

Data types built into OPC UA 2
Fuzzing with AFL 3
Testing data handling functions with libfuzzer 4
Example of fuzzing using libfuzzer 5
Conclusion 6

In an article published in May 2018, we described our approaches to searching for vulnerabilities in industrial systems based on the OPC UA protocol.

Two years later, the issue of ensuring the security of industrial systems based on that protocol is as relevant as ever. Large vendors of industrial software continue to develop and support their products based on implementations of the protocol stack written in C and C++ – languages that have memory use security issues.

We would like to add new information to our security analysis of applications based on the OPC UA protocol. In this article, we:

- Examine new techniques that can be used to search for memory corruption vulnerabilities if the source code is available.
- Discuss an example of fuzzing using libfuzzer. After selecting a test server implementation provided together with the UA ANSI C STACK from OPC Foundation as a vulnerability search target, we demonstrate that this technique makes it sufficiently easy to identify a vulnerability in the server implementation.

We hope that this article will be of use to developers of industrial software.

© 2020 AO KASPERSKY LAB

Data types built into OPC UA

In our earlier OPC UA analysis, we determined that the entire cycle of communication between the client and the server is a series of binary messages structured in a certain way.

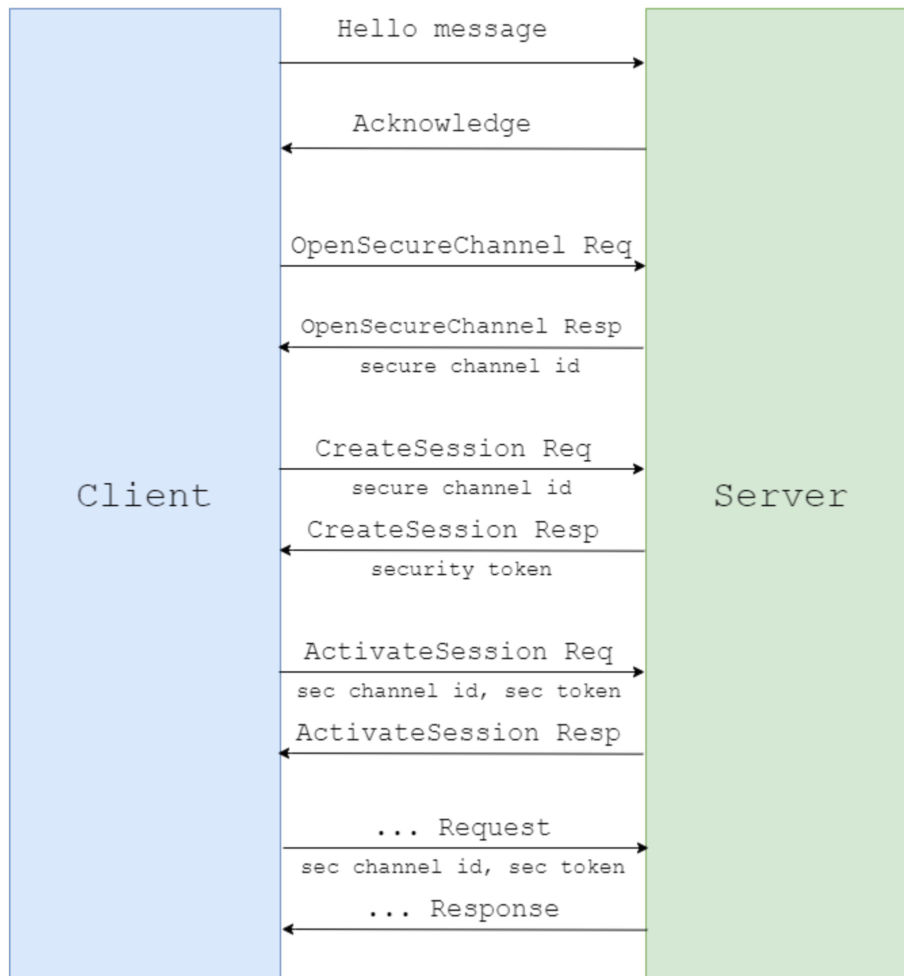

Figure 1. Overview of establishing a connection between a client and an OPC UA based server

We also determined that each application consists of two main parts.

The first part, which is responsible for network communication and initial processing of data received over the network, is the OPC UA stack.

The second part is one of the request handling functions, which receives a special structure created by the stack after it has determined the message type, checked that the message was formed correctly and the session through which the user is sending the message has not expired.

In most cases, request handling functions are written by a specific product's vendor for each type of message that needs to be supported by the product. In order for these handling functions on the server to be available for client requests, they must be recorded in a structure array containing *OpcUa_ServiceType* type structures, which will be passed to the *OpcUa_Endpoint_Create* function via a pointer, so calling that function will be a starting point for us.

© 2020 AO KASPERSKY LAB

```
140 /*============================================================================
141  * The service dispatch information FindServers service.
142  *===========================================================================*/
143 struct _OpcUa_ServiceType my_FindServers_ServiceType =
144 {
145     OpcUaId_FindServersRequest,
146     OpcUa_Null,
147     OpcUa_Server_BeginFindServers,
148     (OpcUa_PfnInvokeService*)my_FindServers
149 };
238 OpcUa_ServiceType*  UaTestServer_SupportedServices[] =
239 {
240     &OTServer_ServiceGetEndpoints,
241     &ServiceCreateSession,
242     &ActivateSession,
243     &CloseSession,
244     &my_Browse_ServiceType,
245     &my_Read_ServiceType,
246     &my_BrowseNext_ServiceType,
247     &my_FindServers_ServiceType,
248     &dummy_CreateSubscription,
249     OpcUa_Null
250 };
```

Figure 2. List of services available on the sample server

As mentioned above, each specific application has two main components — the stack and handler functions. The stack receives messages over the network, handles each message based on its type, processes data contained in the message and only then passes the data on to the handler function. Sometimes data sent in a message can have a sufficiently complicated structure.

Fuzzing with AFL

First, we will revisit our research of several years ago, when we fuzz-tested a sample server that used the UA ANSI C Stack, and reiterate why we selected fuzzing as the main technique for testing the product.

If you would like to know more about the data types built into OPC UA and how they are parsed, you can search the opcua_types.c file for functions with names that end in "_Decode". The file is so large that GitHub will refuse to let you view it. Searching such a large system for flaws without using modern automated vulnerability search techniques can prove meaningless, since it is hard to read and understand this much code in a reasonably short time. However, manual analysis should not be discarded completely, since you often have to understand the inner workings of an application to write an effective fuzzer or find out why your fuzzer is unable to overcome a certain condition and improve code coverage.

We used the AFL fuzzer to test the entire system, from the initial call of the *recv* function to the moment when the server generates the reply and sends it back to the client. Since after each mutation the new input is processed in a new thread, there is no need to worry about possible memory leaks that may occur in the program. This is a major advantage of fuzzing in a separate process. At the same time, AFL has some shortcomings in this case – for example, it is sufficiently hard to get some built-in data types from the original data using its mutations. Even so, it was one of our best support tools: with its help, the process of fuzzing can be started without much effort.

To launch AFL, the project must be compiled using *afl-gcc* instead of the original compiler, adding the environment variable *AFL_USE_ASAN=1*. After this the project is almost ready for fuzzing. The last step is to use a library developed by us, which we will load into the process being tested using *AFL_PRELOAD* (which is similar to LD_PRELOAD). The library substitutes

© 2020 AO KASPERSKY LAB

networking functions (*socket, connect, send, recv, poll, select*, etc.). For example, when calling the *recv* function, the program will 'think' it has read data from a socket, while in reality a function from our library has been called and has in turn read data from a file. This trick significantly accelerates the process of fuzzing.

Testing data handling functions with libfuzzer

The fuzzing technique described above is sufficiently fast. However, if what needs to be tested is data handling functions written for a specific application, rather than the entire ANSI stack, you would want to skip some stages, including the handshake, opening a secure channel, creating a new session, as well as the handling of all the messages generated during these stages, and to fuzz test these functions directly, it possible.

Below we describe how libfuzzer can help us with this task.

Although in this publication we try to describe everything we do in detail, this is not a libfuzzer tutorial. To those of our readers who are not yet familiar with the tool and would like to learn to use it, we strongly recommend reading about it here and here.

Libfuzzer is different from AFL in essential ways. First of all, it is an in-memory fuzzer, which means that all testing is done in one separate process, which in theory should be faster than creating a new process every time.

To test the handler function, we need a way to create arguments for it from our mutating data. This is where our knowledge of the inner workings of the OPC UA stack will come in handy. Let's have a look inside the *OpcUa_BinaryDecoder_ReadMessage* function, which is located in the *Stack/stackcore/opcua_binarydecoder.c* file.

The *OpcUa_BinaryDecoder_ReadMessage* function accepts three parameters:

1. Input parameter *a_pDecoder*, which contains our data stream.
2. Optional in-out parameter *a_ppMessageType*, in which the user can specify the expected message type or keep the *OpcUa_Null* value, in which case the stack will handle any message type known to it.
3. Output parameter *a_ppMessage*, which will return a pointer to our message.

a_ppMessage has the type *OpcUa_Void***, because it is expected that whoever called the function will cast the argument to the necessary type, which was returned in *a_ppMessageType.*

It can be seen in the message body that first it reads the object type identifier from the *Identifier.Numeric* field of the *OpcUa_NodeId cTypeId* object. If the type read is supported by the stack, the function creates a new object and attempts to handle the remaining part of the data based on the object type by calling the *OpcUa_BinaryDecoder_ReadEncodeable* function in order to populate the fields of the object that has just been created.

Here is our action plan for each iteration of fuzzing in the *LLVMFuzzerTestOneInput* function:

1. Initialize the OPC UA stack if it hasn't been initialized already.
2. Initialize the OpcUa_InputStream structure with mutated data. The structure will be used to create the handler for our data — OpcUa_Decoder pDecoder.
3. Call the ReadMessage field function of our decoder, pDecoder. The function is a pointer to the function discussed above, OpcUa_BinaryDecoder_ReadMessage, because we created the OpcUa_Decoder type object using OpcUa_BinaryDecoder_Create.

 © 2020 AO KASPERSKY LAB

4. If the data was created correctly in accordance with the request format, we will get message type and content as an output of the ReadMessage function.
5. If the message generated is of one of the known types, we call the handler function implemented in the application being analyzed, passing the data obtained by parsing the message to the handler function as arguments.
6. The iteration is completed by releasing the memory used in order to avoid memory leaks.

Example of fuzzing using libfuzzer

To go through the entire process on your own, you can use code that we have posted on GitHub. The OPC UA protocol stack by the OPC Foundation is a cross-platform product and in our previous article we discussed fuzzing under GNU/Linux OS. This time, we suggest trying out a relatively new tool — libfuzzer for Windows.

We tested the *my_Browse* data handling function as an example. A crash was detected a few minutes after starting testing.

It can be seen in the code stored in the repository that we replaced the first two arguments of the *my_Browse* data handling function with values equal to one. We did this because these arguments are not used in the *my_Browse* function and, at the same time, they are pointers. As a consequence, the function checks in the beginning whether these arguments are equal to zero, which is the proper thing to do in thus protocol stack. If one of the arguments was equal to zero, the function would terminate with a return code reporting an error.

The case in which these arguments are used in a specific program should be considered separately. In such cases, using these arguments should be avoided to the extent possible by modifying the source code (in this article, we assume that we have access to the source code of the application being tested) or replacing functions that use these arguments with stubs. The same applies to global variables, which can be used in a data handling function and which could be uninitialized or have incorrect values.

Using stubs and improper modifications of the source code during testing could lead to incorrect results, such as a crash identified during fuzzing that is not reproduced on the actual program, or incomplete coverage. The possibility of this kind of behavior should be kept in mind: it is important to keep track of the coverage and be prepared to address this sort of issue by modifying the changes made to the code and using stub functions properly.

We have shown that to fuzz-test a program, it is often not necessary to model all of its behavior. It is sufficient to use a set of techniques that will essentially fix it with 'duct tape', just enough to enable it to run in some way.

To reproduce our experience and build the project, some modifications first need to be made to it: applying a patch containing the fuzzer to the server (or independently writing a target function for fuzzing), adding flags to compile the program together with the instrumentation provided by ASAN and libfuzzer, and linking the final binary file. The repository contains more detailed descriptions of these operations.

The fuzzing speed will naturally depend on the computer on which you run the fuzzer. However, as mentioned above, even when running the fuzzer on a laptop, a crash was identified after a few minutes.

© 2020 AO KASPERSKY LAB

```
PS C:\Users\█████████\UA-AnsiC-Legacy\build> .\bin\AnsiCServer.exe .\crash
WARNING: Failed to find function "__sanitizer_acquire_crash_state".
WARNING: Failed to find function "__sanitizer_print_stack_trace".
WARNING: Failed to find function "__sanitizer_set_death_callback".
INFO: Seed: 3881906032
INFO: Loaded 1 modules   (22093 inline 8-bit counters): 22093 [005C59A8, 005CAFF5),
INFO: Loaded 1 PC tables (22093 PCs): 22093 [00562788,0058D9F0),
C:\Users\█████████\UA-AnsiC-Legacy\build\bin\AnsiCServer.exe: Running 1 inputs 1 time(s) each.
Running: .\crash
initialized!


FINDSERVERS SERVICE=================================
=================================================================
==15800==ERROR: AddressSanitizer: access-violation on unknown address 0x00000000 (pc 0x63d203f6 bp 0x007bf24c sp 0x007bf21c T0)
==15800==The signal is caused by a READ memory access.
==15800==Hint: address points to the zero page.
    #0 0x63d203f5  (C:\WINDOWS\SYSTEM32\ucrtbased.dll+0x100e03f5)
    #1 0xeb081 in my_FindServers C:\Users\█████████\UA-AnsiC-Legacy\AnsiCSample\ansicservermain.c:508
    #2 0xf6026 in LLVMFuzzerTestOneInput C:\Users\█████████\UA-AnsiC-Legacy\AnsiCSample\ansicservermain.c:1662
    #3 0xd0801 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const *, unsigned int) C:\src\llvm_package_1000-final\llvm-project\compiler-rt\lib\fuzzer\FuzzerLoop.cpp:556
    #4 0xe0d93 in fuzzer::RunOneTest(class fuzzer::Fuzzer *, char const *, unsigned int) C:\src\llvm_package_1000-final\llvm-project\compiler-rt\lib\fuzzer\FuzzerDriver.cpp:293
    #5 0xe53b0 in fuzzer::FuzzerDriver(int *, char ***, int (__cdecl *)(unsigned char const *, unsigned int)) C:\src\llvm_package_1000-final\llvm-project\compiler-rt\lib\fuzzer\FuzzerDriver.cpp:779
    #6 0xb1014 in main C:\src\llvm_package_1000-final\llvm-project\compiler-rt\lib\fuzzer\FuzzerMain.cpp:19
    #7 0x4d8fdd in invoke_main f:\dd\vctools\crt\vcstartup\src\startup\exe_common.inl:78
    #8 0x4d9146 in _scrt_common_main_seh f:\dd\vctools\crt\vcstartup\src\startup\exe_common.inl:288
    #9 0x4d921c in _scrt_common_main f:\dd\vctools\crt\vcstartup\src\startup\exe_common.inl:330
    #10 0x4d9227 in mainCRTStartup f:\dd\vctools\crt\vcstartup\src\startup\exe_main.cpp:16
    #11 0x7724f988  (C:\WINDOWS\System32\KERNEL32.DLL+0x6b81f988)
    #12 0x77eb7083  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2e7083)
    #13 0x77eb7053  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2e7053)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: access-violation (C:\WINDOWS\SYSTEM32\ucrtbased.dll+0x100e03f5)
==15800==ABORTING
```

Figure 3. Crash of the fuzzer

After getting the first crash, we need to verify that this behavior is reproduced not only in our fuzzer but on a real-world system, as well.

It can be seen in the following screenshot that when the server handles the data identified, this also results in a crash.

```
(2f48.41d8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\Users\█████████\UA-AnsiC-Legacy-clean\build\bin\AnsiCServer.exe
eax=0000000b ebx=0000000b ecx=fef52c86 edx=010ad37a esi=0000000b edi=010ad37a
eip=5e5f03d0 esp=02a6ec68 ebp=02a6ec98 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
ucrtbased!strncmp+0x20:
5e5f03d0 0fb60411        movzx   eax,byte ptr [ecx+edx]     ds:002b:00000000=??
0:005> g
(2f48.41d8): Access violation - code c0000005 (!!! second chance !!!)
eax=0000000b ebx=0000000b ecx=fef52c86 edx=010ad37a esi=0000000b edi=010ad37a
eip=5e5f03d0 esp=02a6ec68 ebp=02a6ec98 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
ucrtbased!strncmp+0x20:
5e5f03d0 0fb60411        movzx   eax,byte ptr [ecx+edx]     ds:002b:00000000=??

0:005>
```

Figure 4. Crash of the server

We have demonstrated that libfuzzer can be used to identify, quickly and with minimal effort, a vulnerability in data handling functions written for a specific application.

Conclusion

We hope that we were able to demonstrate in this article how easily vulnerabilities that are critical for this type of system can be identified using modern techniques. We believe that proving the effectiveness of fuzzing is not necessary, since this technique has been proved to be effective by the large number of vulnerabilities identified with its help. However, we cannot fail to notice that many industrial vendors still neglect this practice in their development process. This is why we have been trying to popularize fuzzing as one of the main techniques for testing industrial software for memory corruption flaws.

A next step in this research could be to use structure-aware fuzzing and to write a fuzzer based on libprotbuf-mutator.

P.S. It is worth noting that following our previous research, when we reported vulnerabilities identified in a sample server using the UA ANSI C Stack, which had been added to the official repository, to developers from the OPC Foundation, they responded that the vulnerabilities were not critical and did not affect the main result of development conducted by the OPC Foundation. All they did was add a warning to the code to the effect that the server was not a final product. They did fix the vulnerabilities, however.

© 2020 AO KASPERSKY LAB

Kaspersky Industrial Control Systems Cyber Emergency Response Team (Kaspersky ICS CERT) is a global project of Kaspersky aimed at coordinating the efforts of automation system vendors, industrial facility owners and operators, and IT security researchers to protect industrial enterprises from cyberattacks. Kaspersky ICS CERT devotes its efforts primarily to identifying potential and existing threats that target industrial automation systems and the industrial internet of things.

Kaspersky ICS CERT ics-cert@kaspersky.com

© 2020 AO KASPERSKY LAB