Практический пример фаззинга приложений OPC UA

Павел Черемушкин

19.10.2020 Version 1.0

Содержание

Встроенные в OPC UA типы данных ... 2
Метод фаззинга с использованием фаззера AFL ... 3
Тестирование функций обработки с помощью libfuzzer ... 4
Пример фаззинга с использованием libfuzzer ... 5
Заключение ... 7

В статье, опубликованной в мае 2018 года, мы рассказали о наших подходах к поиску уязвимостей в индустриальных системах на базе протокола OPC UA.

Спустя два года вопрос безопасности индустриальных систем на базе данного протокола стоит всё так же остро. Крупные производители индустриального ПО продолжают разработку и поддержку своих продуктов на базе реализаций стека протоколов, которые написаны на небезопасных с точки зрения использования памяти языках программирования С и С++.

Мы хотели бы дополнить информацию об анализе безопасности приложений на базе протокола OPC UA. В этой статье мы:

- Рассматриваем новые методы поиска уязвимостей порчи памяти при наличии исходного кода.
- Приводим пример фаззинга с использованием libfuzzer. Избрав в качестве цели для поиска уязвимостей реализацию тестового сервера, который был предоставлен вместе со стеком UA ANSI C STACK от OPC Foundation, мы покажем, что данным способом можно довольно легко обнаружить уязвимость в реализации сервера.

Мы надеемся, что данная статья будет полезна производителям индустриального программного обеспечения.

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»

Встроенные в OPC UA типы данных

В нашем предыдущем исследовании OPC UA мы выяснили, что весь цикл общения между клиентом и сервером состоит из передачи сообщений бинарного формата определённой структуры.

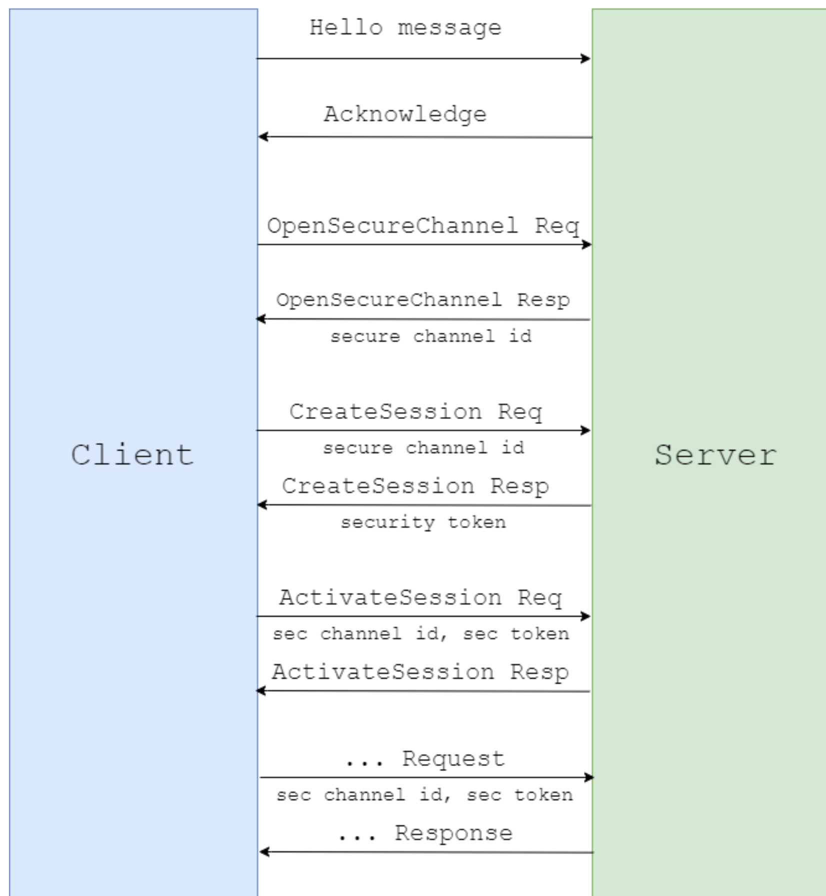

Рис. 1. общая схема установки соединения клиента и сервера на базе OPC UA

Мы также выяснили, что каждое приложение состоит из двух основных частей.

Первой частью, которая отвечает за сетевое взаимодействие и первичную обработку данных, полученных по сети, является непосредственно стек OPC UA.

Второй частью является одна из функций обработки запроса, которая получает специальную структуру, сформированную стеком после того, как тот определил тип сообщения, проверил, что оно правильно сформировано, а сессия, через которую пользователь передаёт сообщение, не истекла.

Функции обработки запроса, как правило, пишутся самим производителем конкретного продукта для каждого типа сообщений, которые необходимо поддерживать в продукте. Для того чтобы данные функции обработки на сервере были доступны для обращения клиента, они должны быть записаны в массив структур типа *OpcUa_ServiceType[]*, который будет передан по указателю в функцию *OpcUa_Endpoint_Create*, поэтому вызов данной функции будет служить для нас отправной точкой.

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»

```
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 };
```

Рис. 2. Список доступных сервисов из сервера-примера

Как уже было сказано, каждое отдельное приложение состоит из двух основных компонентов: стека и функций обработчиков. Стек принимает сообщения по сети, обрабатывает каждое сообщение в соответствии с его типом, обрабатывает данные, содержащиеся в нём, и только потом передаёт их далее в функцию-обработчик. Данные, переданные в сообщении, имеют порой довольно сложную структуру.

Метод фаззинга с использованием фаззера AFL

Для начала напомним, каким образом несколько лет назад мы проводили тестирование примера сервера, использующего UA ANSI C Stack, при помощи фаззинга, а также почему мы выбрали именно фаззинг как основной метод тестирования данного продукта.

Если вам захочется больше узнать про встроенные в OPC UA типы данных и о том, как происходит их парсинг, вы можете поискать функции, имена которых заканчиваются на `*_Decode*` в файле *opcua_types.c*. Данный файл является настолько большим, что GitHub откажется вам его показывать. Поиск ошибок в настолько большой системе без использования современных методов автоматического обнаружения уязвимостей может оказаться бессмысленным, так как прочитать и понять такой объем кода за разумное время сложно. Однако не следует полностью исключать ручной анализ, так как зачастую для того, чтобы написать эффективный фаззер или понять, почему фаззер не может преодолеть определенное условие и увеличить покрытие кода, необходимо понимать, как устроено приложение изнутри.

В том, чтобы исследовать всю систему от первого вызова функции *recv* до момента, когда сервер формирует ответ и отправляет его обратно клиенту, нам помог фаззер AFL. Так как после каждой мутации новые входные данные обрабатываются в новом потоке, то нас не волнуют утечки памяти, которые могу возникать в программе. Это является большим достоинством фаззинга в отдельном процессе. Хотя AFL в данном случае имеет и минусы – например, при помощи его мутаций довольно тяжело получить некоторые встроенные типы данных из исходных. Несмотря на это, он являлся для нас одним из лучших вспомогательных инструментов – начать процесс фаззинга с ним можно без больших трудозатрат.

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»

Для того чтобы запустить AFL, необходимо скомпилировать проект при помощи *afl-gcc,* вместо исходного компилятора, добавив переменную окружения *AFL_USE_ASAN=1,* и проект практически готов к фаззингу. Последним шагом будет использование нашей библиотеки, которую мы загрузим в тестируемый процесс при помощи *AFL_PRELOAD* (аналог LD_PRELOAD). Данная библиотека подменяет функции по работе с сетью (*socket, connect, send, recv, poll, select*, etc.). То есть, например, при вызове функции *recv* программа будет думать, что считала данные из сокета, хотя была вызвана функция из нашей библиотеки, которая считала данные из соответствующего файла. Этот трюк значительно убыстряет процесс фаззинга.

Тестирование функций обработки с помощью libfuzzer

Описанный выше метод фаззинга работает достаточно быстро. Однако, если стоит задача тестирования не ANSI stack, а функций обработки данных, написанных для конкретного приложения, хотелось бы пропустить этапы обмена приветственными сообщениями, открытие безопасного канала, создание новой сессии и этап обработки всех этих сообщений и фаззить эти функции напрямую, если такое возможно.

Рассмотрим, как нам может помочь инструмент libfuzzer в решении данной задачи.

Несмотря на то, что в данной публикации мы стараемся очень подробно описывать все наши действия, она не является обучающим материалом по использованию libfuzzer. Поэтому, если читатель ещё не знаком с данным инструментом и желает обучиться его использованию, мы настоятельно рекомендуем прочесть о нём здесь и здесь.

Libfuzzer имеет принципиальные отличия от AFL. В первую очередь, это in-memory фаззер, то есть всё тестирование происходит в одном отдельном процессе, что, по идее, должно быть быстрее, чем каждый раз запускать отдельный процесс.

Чтобы протестировать функцию-обработчик, нам необходимо каким-то образом составить для неё аргументы из наших мутирующих данных. В этом нам помогут наши знания внутренностей стека OPC UA. Заглянем внутрь функции *OpcUa_BinaryDecoder_ReadMessage*, которая находится в файле *Stack/stackcore/opcua_binarydecoder.c*.

Функция *OpcUa_BinaryDecoder_ReadMessage* принимает три параметра:

1. Входной параметр *a_pDecoder*, который содержит внутри себя поток с нашими данными.
2. Необязательный in-out параметр *a_ppMessageType*, в котором пользователь может указать ожидаемый тип сообщения, а может оставить его равным значению *OpcUa_Null*, чтобы стек обработал любой известный ему тип сообщения.
3. Выходной параметр *a_ppMessage*, который вернёт указатель на наше сообщение.

a_ppMessage имеет тип *OpcUa_Void***, потому что ожидается, что тот, кто вызвал данную функцию, приведёт этот аргумент к нужному типу, который вернулся в *a_ppMessageType*.

В теле функции можно увидеть, что сначала она считывает идентификатор типа объекта, который находится в поле *Identifier.Numeric* объекта *OpcUa_NodeId cTypeId*. Если прочитанный тип поддерживается стеком, то функция создаёт новый объект и пытается обработать оставшуюся часть данных в соответствии с типом объекта

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»

с помощью вызова функци *OpcUa_BinaryDecoder_ReadEncodeable*, чтобы заполнить поля только что созданного объекта.

Изложим план наших дальнейших действий на каждой итерации фаззинга в функции *LLVMFuzzerTestOneInput*:

1. Инициализируем OPC UA стек, если он не был инициализирован.
2. Инициализируем мутировавшими данными структуру OpcUa_InputStream, которая будет использована для создания обработчика наших данных — OpcUa_Decoder pDecoder.
3. У нашего обработчика pDecoder мы вызовем функцию-поле ReadMessage, которая является указателем на рассмотренную нами функцию OpcUa_BinaryDecoder_ReadMessage, так как объект типа OpcUa_Decoder создан нами при помощи OpcUa_BinaryDecoder_Create.
4. Если данные были правильно сформированы согласно формату запроса, то на выходе из функции ReadMessage мы получим тип сообщения и его содержимое.
5. Если сообщение получилось одного из известных нам типов, мы совершим вызов функции-обработчика, имплементированной в самом приложении, и передадим ей в качестве аргументов полученные нами в результате парсинга сообщения данные.
6. Завершим итерацию тем, что освободим использованную нами память, чтобы избежать утечек.

Пример фаззинга с использованием libfuzzer

Чтобы проделать весь процесс, вы можете использовать код, который мы выложили на GitHub. Стек протоколов OPC UA от OPC Foundation является кроссплатформенным продуктом, и в прошлой статье мы проводили фаззинг на операционной системе GNU/Linux. В данной статье предлагается опробовать сравнительно новый инструмент – libfuzzer для ОС Windows.

Для примера мы протестировали функцию обработки данных *my_Browse.* Падение было обнаружено спустя несколько минут после начала тестирования.

Как можно заметить в коде в репозитория, мы заменили первые два аргумента функции обработки данных *my_Browse*, на единицы. Это сделано потому, что данные аргументы не используются в функции *my_Browse,* при этом они являются указателями. Следственно, в самом начале функции, как это положено в данном стеке протоколов, они проверяются на равенство нулю. Если бы один из данных аргументов был равен нулю, то функция сразу бы завершилась и вернулась с кодом возврата, который сообщает об ошибке.

Отдельно необходимо рассмотреть случай, когда в конкретной программе используются данные аргументы. В таких случаях необходимо избежать, насколько это возможно, использования этих аргументов путём модификации исходного кода (в данной статье мы исходим из предположения, что исходный код тестируемого приложения имеется в нашем распоряжении) или замены функций, которые их используют, на «заглушки». То же самое относится и к глобальным переменным, которые могут использоваться в функции обработки данных, при этом они могут быть не инициализированными или иметь неправильное значение.

Из-за использования «заглушек» и неправильных модификаций исходного кода в ходе тестирования можно получить неправильные результаты: например, в ходе фаззинга было найдено падение программы, которое не воспроизводится на реальной

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»

программе, или покрытие может оказаться неполным. Необходимо иметь ввиду возможность такого поведения, поэтому в ходе фаззинга обязательно нужно следить за покрытием и быть готовым решать подобного рода проблемы корректировкой внесённых в код исправлений и правильным использованием функций-заглушек.

Итак, чтобы провести фаззинг-тестирование программы, у нас зачастую нет необходимости полностью моделировать её поведение. Достаточно использовать набор приёмов, который поможет нам запустить программу хоть как-то, «на костылях».

Для того чтобы повторить наш опыт и собрать проект, необходимо предварительно внести в него некоторые изменения: применить к серверу-примеру патч, который содержит в себе фаззер (или написать target функцию для фаззинга самостоятельно), добавить флаги компиляции программы вместе с инструментацией, которую предоставляют ASAN и libfuzzer, слинковать финальный бинарный файл. Более подробно данные действия описаны в самом репозитории.

Естественно, скорость фаззинга будет зависеть от компьютера, на котором вы запускаете фаззер. Однако даже в случае запуска данного фаззера на ноутбуке, как уже было сказано выше, падение было обнаружено спустя несколько минут.

```
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
```

Рис. 3. Падение фаззера

После получения первого падения необходимо проверить, что данное поведение воспроизводится не только в нашем фаззере, но и на рабочей системе.

Как можно видеть на следующем скриншоте, обработка сервером найденных данных также приводит к падению.

```
(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>
```

Рис. 4. Падение сервера

Таким образом, используя libfuzzer, можно без особых трудозатрат довольно быстро обнаружить уязвимость в функциях обработки данных, написанных для конкретного приложения.

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»

Заключение

Мы надеемся, что в данной статье нам удалось показать, насколько просто, используя современные методы, можно найти критические для данного рода систем уязвимости. Мы не считаем, что должны доказывать эффективность тестирования методом фаззинга, так как эффективность этой методики уже доказана множеством найденных уязвимостей. Однако мы не можем не замечать, что некоторые индустриальные вендоры в ходе разработки по-прежнему пренебрегают данной практикой. Поэтому мы всячески пытаемся популяризовать фаззинг как один из основных методов тестирования индустриального ПО с целью поиска ошибок порчи памяти.

Следующим шагом в развитии данного исследования может стать использование structure aware фаззинга и написание фаззера на основе libprotbuf-mutator.

PS. Следует сказать, что, когда мы в прошлый раз сообщили о найденных уязвимостях в примере сервера, использующего UA ANSI C Stack, который был добавлен в официальный репозиторий, разработчикам из OPC Foundation, они заявили, что найденные уязвимости не являются критичными и влияющими на основной результат разработки OPC Foundation. Они добавили в код лишь предупреждение, что данный сервер не является конечным продуктом. Однако сами уязвимости всё-таки были исправлены.

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»

Центр реагирования на инциденты информационной безопасности промышленных инфраструктур «Лаборатории Касперского» (Kaspersky ICS CERT) — глобальный проект «Лаборатории Касперского», нацеленный на координацию действий производителей систем автоматизации, владельцев и операторов промышленных объектов, исследователей информационной безопасности при решении задач защиты промышленных предприятий и объектов критически важных инфраструктур.

Kaspersky ICS CERT ics-cert@kaspersky.com

© 2020 АО «ЛАБОРАТОРИЯ КАСПЕРСКОГО»