KASPERSKY lab

Исследование безопасности OPC UA

Павел Черемушкин

Сергей Темников

Kaspersky Lab ICS CERT

Содержание

В этой статье мы рассказываем про наш проект поиска уязвимостей в реализациях протокола OPC UA. Публикуя данный материал, мы хотим обратить внимание производителей программного обеспечения для систем промышленной автоматизации и промышленного интернета вещей на обнаруженные нами и оказавшиеся типичными проблемы в разработке продуктов с использованием подобных общедоступных технологий. Мы надеемся, что статья поможет производителям ПО добиться лучшей защищённости их продуктов от современных компьютерных атак. Мы рассказываем также о некоторых наших технических приёмах и находках, которые, возможно, будут полезны производителям ПО для контроля качества выпускаемых продуктов, а также другим исследователям безопасности программного обеспечения.

Почему мы выбрали протокол OPC UA в качестве объекта исследования

Стандарт IEC 62541 OPC Unified Architecture (OPC UA) разработан в 2006 году консорциумом OPC Foundation для надёжной и, что немаловажно, безопасной передачи данных между различными системами в технологической сети. Стандарт является усовершенствованной версией своего предшественника – протокола OPC, повсеместно применяемого на современных промышленных объектах.

Системы мониторинга и управления, реализованные на продуктах разных производителей, нередко используют несовместимые, часто закрытые, протоколы сетевой коммуникации. Шлюзы/серверы OPC являются связующим звеном между различными системами автоматизированного управления технологическим процессом и системами телеметрии, мониторинга и телеуправления, позволяя унифицировать процессы управления промышленными предприятиями.

Будучи реализована на технологии Microsoft DCOM, предыдущая версия протокола обладала рядом присущих этой технологии существенных ограничений. Чтобы уйти от ограничений технологии DCOM и решить некоторые другие обнаруженные за время использования протокола OPC проблемы, OPC Foundation разработал и выпустил новую версию протокола.

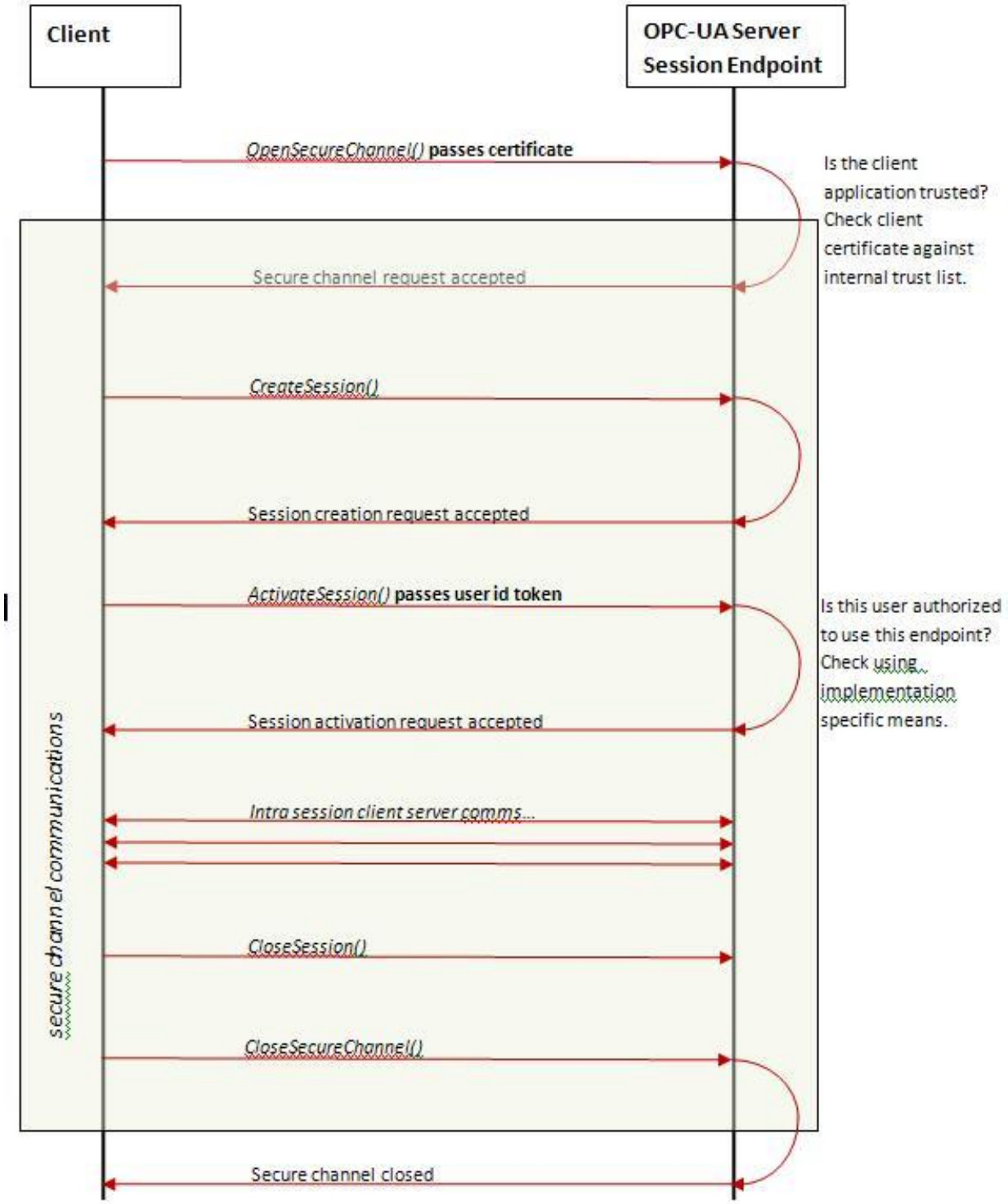
Благодаря своим новым свойствам и продуманной архитектуре протокол OPC UA стремительно набирает популярность среди производителей систем автоматизации, шлюзы OPC UA активно внедряются на промышленных предприятиях по всему миру. Протокол становится основой коммуникаций между компонентами систем промышленного интернета вещей и умных городов.

Безопасность технологий, которые используются многими разработчиками систем автоматизации и имеют потенциал очень широкого использования на промышленных объектах по всему миру, является одним из приоритетных направлений исследований Kaspersky Lab ICS CERT. Поэтому мы и занялись исследованием OPC UA.

Другой причиной стал тот факт, что «Лаборатория Касперского» является членом консорциума OPC Foundation, и мы чувствуем свою ответственность за безопасность разрабатываемого в рамках консорциума технологий. Забегая вперёд, скажем, что по результатам наших исследований мы получили приглашение вступить в OPC Foundation Security Working Group и с благодарностью его приняли.

Протокол OPC UA

Изначально OPC UA разрабатывали так, чтобы он мог поддерживать два типа передаваемых данных: традиционный (для предыдущих версий стандарта) бинарный формат и SOAP/XML. На сегодняшний день в мире IT передача данных в формате SOAP/XML считается устаревшей и почти не используется в современных продуктах и решениях. Перспективы его широкого применения в системах промышленной автоматизации в будущем пока туманны, поэтому мы приняли решение сфокусировать своё исследование именно на бинарном формате.



Источник: https://readthedocs.web.cern.ch/download/attachments/21178021/OPC-UA-Secure-Channel.JPG?version=1&modificationDate=1286181543000&api=v2

Перехватив пакеты, которыми обмениваются запущенные на хосте сервисы, можно легко понять их структуру. Существует четыре вида сообщений, которые передаются по протоколу OPC UA:

- HELLO
- OPEN
- MESSAGE
- CLOSE

Первым сообщением всегда является HELLO (HEL), которое служит маркером начала передачи данных между клиентом и сервером. В ответ сервер посылает клиенту сообщение ACKNOWLEDGE (ACK). После первичного обмена сообщениями клиент обычно посылает сообщение OPEN, которое обозначает открытие канала передачи данных с предложенным клиентом методом шифрования данных. В ответ сервер отправляет сообщение OPEN (OPN), которое содержит уникальный ID канала передачи данных, а также показывает, что он согласен на предложенный метод шифрования (или его отсутствие).

Теперь клиент и сервер могут начинать обмен сообщениями MESSAGE (MSG), каждое из которых содержит ID канала передачи данных, тип запроса или ответа, временную метку, массивы передаваемых данных и прочее. В конце сессии передачи данных посылается сообщение CLOSE (CLO), после чего соединение обрывается.

OPC UA – стандарт, у которого есть множество реализаций. В нашем исследовании мы рассматривали только конкретную реализацию протокола, разработанную консорциумом OPC Foundation.

Первый этап

Наш интерес к исследованию протокола OPC UA впервые возник в ходе проведения командой Kaspersky Lab ICS CERT аудитов безопасности и тестов на проникновение на нескольких промышленных предприятиях. На всех этих предприятиях использовался один и тот же программный продукт для управления технологическим процессом (ICS). По согласованию с заказчиками в рамках теста на проникновение мы занялись поиском уязвимостей в этом программном продукте.

Оказалось, что часть сетевых сервисов изучаемой системы взаимодействуют по протоколу OPC UA, а большинство исполняемых файлов используют динамическую библиотеку `uastack.dll`.

Взявшись за исследование безопасности реализации протокола, в первую очередь мы решили разработать базовый «глупый» фаззер, основанный на мутациях.

«Глупый» фаззинг, несмотря на своё название, часто бывает весьма полезен и способен в ряде случаев существенно повысить вероятность нахождения уязвимостей. Разработка «умного» фаззера для конкретной программы с учетом принципов и алгоритмов её работы – процесс трудоёмкий. А «глупый» фаззер помогает быстро обнаружить тривиальные уязвимости, до которых бывает сложно добраться руками, особенно когда объем кода анализируемого ПО весьма велик – как было в нашем случае.

Архитектура OPC UA Stack делает фаззинг функций напрямую в памяти процесса трудоёмкой задачей. Для того чтобы функции, которые мы хотим проверить на предмет уязвимостей, правильно работали, в процессе фаззинга необходимо передавать правильно сформированные аргументы функции и инициализировать глобальные переменные, которые являются большими по количеству полей структурами. Мы решили не прибегать к фаззингу функций напрямую из памяти. Написанный нами фаззер общался с изучаемым приложением по сети.

Алгоритм фаззера имел следующий вид:

- считать входные последовательности данных
- преобразовать их псевдослучайным образом
- послать на вход программе по сети
- принять ответ от сервера
- повторить

Написав базовый набор мутаций (bitflip, byteflip, арифметические мутации, подставить магическое число, обнулить последовательность данных, подставить длинную последовательность данных), нам удалось найти первую уязвимость в uastack.dll. Это была уязвимость типа порчи памяти в результате переполнения на куче (heap corruption), успешная эксплуатация которой может давать злоумышленнику возможность удалённого исполнения кода (RCE), в данном случае – с правами NT AUTHORITY/SYSTEM. Обнаруженная уязвимость заключалась в том, что в функции обработки данных, только что считанных из сокета, неправильно подсчитывался размер данных, которые далее копировались в буфер, созданный на куче.

При внимательном рассмотрении удалось обнаружить, что версия библиотеки uastack.dll, содержащая уязвимость, была скомпилирована разработчиками продукта. По всей видимости, уязвимость была привнесена в код в процессе его модификации. Найти эту уязвимость в версиях библиотеки от OPC Foundation не удалось.

Вторая уязвимость была найдена в приложении, написанном на .NET, которое использовало UA .NET Stack. В ходе анализа трафика приложения в wireshark мы заметили, что в диссекторе у некоторых пакетов есть битовое поле is_xml, в котором стоял 0 в качестве значения. Проанализировав приложение, мы обнаружили, что оно использует функцию XmlDocument, которая для .NET версии 4.5 и ранее была уязвима к атаке XXE. То есть, если поменять битовое поле is_xml с 0 на 1 и добавить в тело запроса специальным образом сформированный XML-пакет (атака XXE), мы получим возможность удаленного чтения файла (out-of-bound file read) с правами NT AUTHORITY/SYSTEM, а при некоторых условиях и возможность удаленного исполнения кода (RCE).

Данное приложение, судя по метаданным, хотя и входило в набор ПО рассматриваемой ICS, было разработано не вендором, а консорциумом OPC Foundation, и являлось обычным discovery сервером. Это означает, что и другие продукты, которые используют технологию OPC UA от OPC Foundation, могут содержать данный сервер и будут уязвимы к XXE атаке, что делает эту уязвимость гораздо более ценной с точки зрения атакующего.

Так мы сделали первый шаг в нашем исследовании, по результатам которого приняли решение продолжить изучение реализации OPC UA от консорциума OPC Foundation, а также продуктов, которые его используют.

Исследование OPC UA

Для того чтобы найти уязвимости в реализации протокола OPC UA от консорциума OPC Foundation, необходимо исследовать:

- OPC UA Stack (ANSI C, .NET, JAVA);
- Приложения OPC Foundation, которые используют OPC UA Stack (такие как упомянутый выше OPC UA .NET Discovery Server);
- Приложения прочих разработчиков ПО, которые используют OPC UA Stack.

В ходе исследования мы поставили перед собой задачу найти оптимальные способы поиска уязвимостей во всех трёх категориях.

Фаззинг UA ANSI C Stack

Тут сразу же следует отметить наличие проблемы с поиском уязвимостей в OPC UA Stack. Разработчики OPC Foundation предоставляют библиотеки, которые являются набором экспортируемых функций наподобие API, соответствующих спецификации. В таких случаях часто сложно определить, является ли обнаруженная потенциальная проблема безопасности на самом деле уязвимостью. Для однозначного ответа на этот вопрос необходимо понимать, как и для чего используется потенциально уязвимая функция – т.е. нужен пример программы, использующей данную библиотеку. В нашем случае было трудно судить об уязвимостях в OPC UA Stack в отрыве от реализаций конечного приложения.

В решении этой проблемы с поиском уязвимостей нам помог открытый код, размещенный в репозитории OPC Foundation на GitHub, в котором присутствует пример сервера, использующего UA ANSI C Stack. В ходе исследования компонентов ICS нам не так часто удается получить исходный код продуктов. Большинство приложений ICS – коммерческие продукты, разработанные чаще всего для ОС Windows и выпускаемые с лицензионным соглашением, которое не предполагает открытие исходного кода. В нашем случае исследование открытого кода помогло обнаружить ошибки как в самом сервере, так и в библиотеке. Исходный код UA ANSI C Stack пригодился для ручного анализа кода и фаззинга, он также помог понять, добавлена ли вендором новая функциональность в конкретную реализацию UA ANSI C Stack.

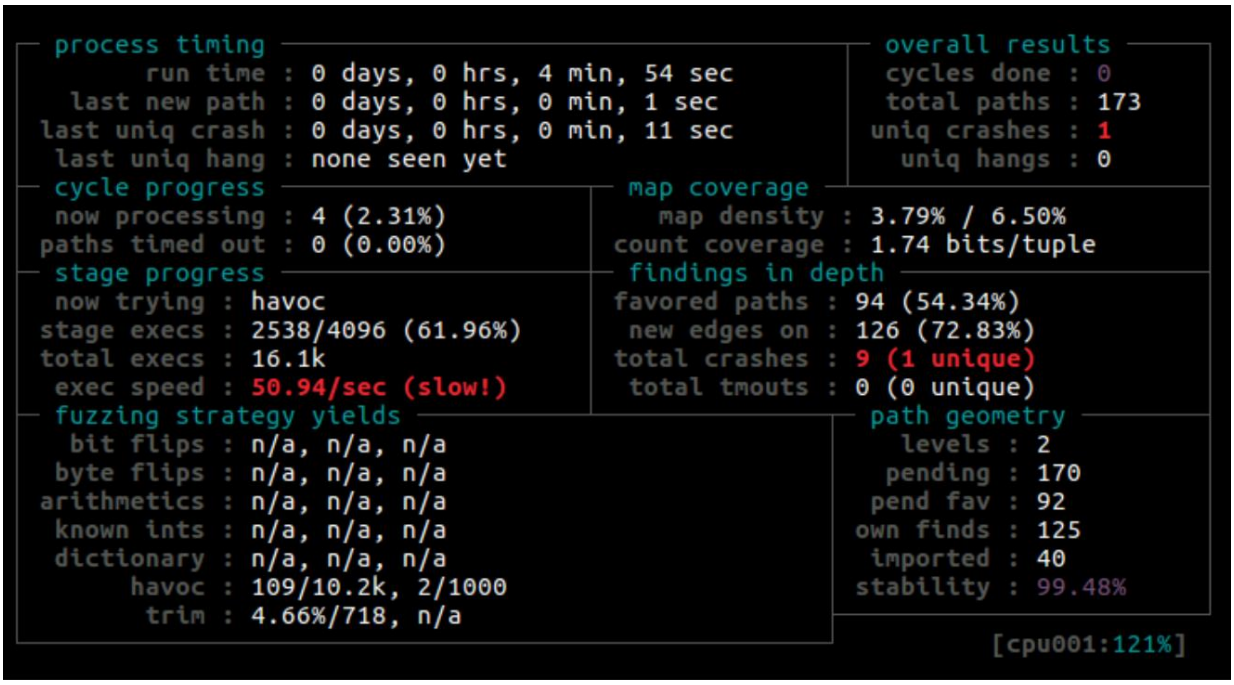
UA ANSI C Stack (как и практически все продукты консорциума OPC UA) позиционируется не только как безопасное, но и как кроссплатформенное решение. Это нам очень помогло в ходе фаззинга, так как мы смогли собрать UA ANSI C Stack вместе с примером кода сервера, которые разработчики опубликовали в своём GitHub аккаунте, на Linux-системе с бинарной инструментацией исходного кода, и провести его фаззинг при помощи AFL.

Для того чтобы ускорить фаззинг, мы перегрузили функции по работе с сетью socket/sendto/recvfrom/accept/bind/select/... так, что вместо работы с сетью они считывали входные данные из локального файла, а также скомпилировали нашу программу с AddressSanitizer.

Для формирования исходного набора примеров мы использовали тот же метод, что и для нашего первого «глупого» фаззера, а именно захват трафика, который идёт к сервису от произвольного клиента при помощи tcpdump. Мы также добавили в наш фаззер несколько усовершенствований – словарь, сформированный исключительно для OPC UA, и специальные мутации.

Из спецификации бинарного формата передачи данных в OPC UA следует, что для AFL достаточно сложно перейти, к примеру, от бинарной репрезентации пустой строки в OPC UA ("\xff\xff\xff\xff") до строки, содержащей 4 случайных байта (например, "\x04\x00\x00\x00AAAA"). Поэтому мы реализовали собственный механизм мутации, который работал с внутренними структурами OPC UA и изменял их в зависимости от типа.

Собрав все усовершенствования нашего фаззера, в течение нескольких минут мы получили первое падение программы.



Проанализировав дампы памяти процесса в момент падения, мы обнаружили в UA ANSI C Stack уязвимость, эксплуатация которой может привести как минимум к DoS.

Фаззинг приложений от OPC Foundation

На предыдущем этапе мы уже провели фаззинг UA ANSI C Stack и примера приложения от OPC Foundation, поэтому при исследовании готовых продуктов консорциума мы хотели избежать повторного тестирования OPC UA Stack и проводить фаззинг только отдельных компонентов, написанных поверх стека. Для этого нам понадобилось знание архитектуры OPC UA, а также знание о том, чем приложения, использующие OPC UA Stack, отличаются друг от друга.

Двумя основными функциями в любом приложении, использующем OPC UA Stack, являются функции OpcUa_Endpoint_Create и OpcUa_Endpoint_Open. Первая указывает приложению возможные типы каналов обмена данными между сервером и клиентом, а также список доступных сервисов. Функция OpcUa_Endpoint_Open определяет, из какой сети будет доступен сервис, а также какие режимы шифрования будут на нём доступны.

Список доступных сервисов задаётся при помощи таблицы сервисов, в которой перечислены структуры данных с информацией непосредственно о каждом сервисе. В каждой такой структуре содержатся данные о типе поддерживаемого запроса, типе ответа, а также две callback-функции, которые будут вызваны в ходе пре-обработки и пост-обработки запроса (функции пре обработки чаще всего являются «заглушками»). В функцию пре-обработки запросов мы поместили код преобразователя, который использует мутированные данные на входе, а на выходе выдает правильно сформированную структуру, соответствующую типу запроса. Таким образом нам удалось миновать этап старта приложения, запуск event-loop для создания отдельного потока чтения из нашего псевдо-сокета и т.д., что позволило ускорить наш фаззинг с 50 exec/s до 2000 exec/s.

В результате работы усовершенствованного таким образом «глупого» фаззера нами было найдено ещё 8 уязвимостей в приложениях от OPC Foundation.

Исследование сторонних приложений, использующих OPC UA Stack

Завершив этап исследования продуктов OPC Foundation, мы перешли к изучению коммерческих продуктов, использующих OPC UA Stack. Из систем ICS, встретившихся нам в процессе проведения тестов на проникновение и исследования состояния защищённости объектов нескольких наших заказчиков, мы выбрали несколько продуктов от различных вендоров, включая продукты мировых лидеров этой отрасли. Согласовав работы с заказчиками, мы приступили к исследованию реализации протокола OPC UA в этих продуктах.

В поиске бинарных уязвимостей фаззинг – один из самых эффективных методов. В предыдущих случаях, когда мы исследовали продукты на Linux-системе, мы воспользовались методами бинарной инструментации исходного кода и фаззером AFL. Исследуемые нами коммерческие продукты, использующиеOPC UA Stack, разработаны под ОС Windows, для которой есть аналог фаззера AFL под названием WinAFL. По сути, WinAFL – это портированный на Windows фаззер AFL. Однако, ввиду различий между ОС, в нём присутствует ряд существенных изменений. Вместо системных вызовов из ядра Linux он использует функции WinAPI, а вместо статической инструментации исходного кода – динамическую инструментацию бинарных файлов DynamoRIO. В сумме эти отличия существенно снижают производительность WinAFL по сравнению с AFL.

Для того чтобы работать с WinAFL по общей схеме, необходимо написать программу, которая будет считывать из специально созданного файла данные и вызывать функцию из исполняемого файла или библиотеки. Тогда WinAFL зациклит этот процесс при помощи бинарной инструментации и будет вызывать эту функцию много раз, получая отклик от запущенной программы и перезапуская функцию с мутированными данными в качестве аргументов. Это позволяет не перезапускать программу каждый раз на новых входных данных, что хорошо, так как создание нового процесса в ОС Windows занимает много процессорного времени.

К сожалению, данная схема фаззинга для нашего случая не подходила. В силу особенностей асинхронной архитектуры OPC UA Stack обработка данных, получаемых и передаваемых по сети, реализуется в call-back-функциях. Поэтому нельзя для каждого типа запросов выделить одну функцию обработки данных, которой можно было бы передать на вход указатель на буфер с данными и их размер, как это требуется фаззером WinAFL.

В исходном коде фаззера WinAFL мы обнаружили комментарии касательно фаззинга сетевых приложений, оставленные разработчиком. Мы последовали его рекомендациям по реализации сетевого фаззинга, но внесли в них изменения. Мы добавили функциональность взаимодействия с локальным сетевым приложением непосредственно в код фаззера. В результате вместо того, чтобы запускать программу, фаззер посылает полезную нагрузку по сети приложению, которое уже запущено в этот момент под DynamoRIO.

Однако, при всех наших усилиях, нам удалось добиться скорости фаззинга всего в 5 exec/s. Это настолько медленная скорость, что для того, чтобы найти уязвимость даже таким умным фаззером, как AFL, потребовалось бы слишком много времени.

Поэтому мы решили вернуться к нашему «глупому» фаззеру и усовершенствовать его.

1. Мы усовершенствовали механизм мутаций, изменив алгоритм генерации данных с учётом наших знаний о типах передаваемых в OPC UA Stack данных.
2. Мы создали набор примеров для каждого из поддерживаемых сервисов (в этом нам очень помогла библиотека python-opcua, которая имеет функции взаимодействия практически со всеми возможными сервисами OPC UA).
3. При использовании фаззера с динамической бинарной инструментацией для тестирования многопоточных приложений, подобных нашему, задача поиска новых ветвей в коде приложения – нетривиальна, ведь сложно определить, какие же именно данные на входе привели к тому или иному поведению приложения. Поскольку наш фаззер общался с приложением по сети, и мы могли однозначно связывать ответ сервера с переданными ему данными (поскольку коммуникации происходят в рамках одной сессии), нам эту проблему решать не пришлось. Мы реализовали алгоритм, который определял, что удалось обнаружить новый путь выполнения программы просто по получению нового, ранее не наблюдаемого, отклика от сервера.

В результате описанных улучшений наш «глупый» фаззер стал не таким уж и «глупым», и количество исполнений в секунду выросло с 1-2 до 70, что является хорошим показателем для сетевого фаззинга. С его помощью мы обнаружили ещё две новые уязвимости, которые нам не удавалось обнаружить при помощи умного фаззинга.

Результаты

На конец марта 2018 года итогами нашего исследования стали 17 найденных и закрытых уязвимостей нулевого дня в продуктах OPC Foundation и несколько уязвимостей – в коммерческих приложениях, которые их используют.

Обо всех найденных в процессе исследования уязвимостях мы незамедлительно сообщали разработчикам уязвимого ПО.

На протяжении всего исследования специалисты OPC Foundation и представители команд разработки коммерческих продуктов очень быстро реагировали на присылаемую нами информацию о найденных уязвимостях и своевременно их устраняли.

Большинство ошибок в продуктах сторонних производителей ПО, использующих OPC UA Stack, оказалось связано с тем, что разработчики неправильно использовали функции из предоставляемого OPC Foundation API, реализованного в библиотеке uastack.dll – например, неверно интерпретировали значения полей передаваемых структур данных.

Также мы обнаружили, что в нескольких случаях модификация библиотеки uastack.dll разработчиками коммерческого ПО приводила к уязвимости в продукте. Один из примеров – небезопасная реализация функций чтения данных из сокета в коммерческом продукте. Оригинальная реализация этой функции от OPC Foundation при этом ошибки не содержала. Зачем разработчику коммерческого ПО потребовалось модифицировать логику чтения, нам неизвестно. Однако очевидно, что важности дополнительных проверок, заложенных в реализации от OPC Foundation и обеспечивавших функцию безопасности, разработчик при реализации своего варианта не осознал.

В ходе исследования коммерческого ПО мы также обнаружили, что при его создании разработчики берут код примеров использования OPC UA Stack и копируют его в код своих приложений, понадеявшись на то, что OPC Foundation позаботился о безопасности этих фрагментов кода так же, как и о безопасности кода самой библиотеки. Это их предположение, к сожалению, оказалось неверным.

Эксплуатация некоторых из обнаруженных нами уязвимостей приводит к DoS и к возможности удалённого исполнения кода. Напомним, что в промышленных системах уязвимости типа отказ-в-обслуживании представляют более существенную опасность, чем в каком-либо другом ПО. Отказ в обслуживании систем телеметрии и телеуправления может приводить к финансовым потерям предприятия, а в некоторых случаях даже к нарушениям и остановке технологического процесса. Теоретически возможна даже порча дорогостоящего оборудования и другой физический ущерб.

Заключение

Тот факт, что консорциум OPC Foundation открывает исходный код своих проектов, безусловно, говорит об открытости и желании консорциума сделать свои продукты более безопасными.

С другой стороны, проведённый анализ показал, что текущая реализация OPC UA Stack не только содержит уязвимости, но и не лишена ряда существенных фундаментальных проблем.

Во-первых, описанные выше ошибки разработчиков коммерческого ПО, использующего OPC UA Stack, позволяют предположить, что OPC UA Stack реализован не совсем очевидным для пользователей образом. Анализ исходного кода от OPC UA Stack это предположение, к сожалению, подтверждает. Текущая реализация протокола изобилует разбросанными по коду вычислениями над указателями, использованием небезопасных структур данных, магических констант, копированием кода проверки параметров из функции в функцию и прочими архаизмами, от которых в современных программных продуктах принято избавляться, во многом именно с целью повысить безопасность разработки. Документация кода при этом не отличается подробностью, что повышает вероятность ошибки при его использовании и модификации.

Во-вторых, разработчики OPC UA явно недооценивают степень доверия компаний-разработчиков ПО ко всему коду, предоставляемому консорциумом OPC Foundation. На наш взгляд, оставлять уязвимости в коде примеров использования API – совсем неправильно, даже несмотря на то, что примеры использования API не входят в список сертифицируемых продуктов OPC Foundation.

В-третьих, по нашему мнению, есть проблемы с контролем качества и сертифицированных продуктов OPC Foundation.

Вероятно, разработчики OPC UA не используют технологию фаззинг-тестирования, наподобие описанной в этой статье, в процессе контроля качества своего продукта – об этом говорит статистика обнаруженных нами уязвимостей.

В открытом исходном коде не содержится кода unit- или каких-либо других автоматических тестов, что усложняет тестирование продуктов, использующих OPC UA Stack, в случае модификации кода разработчиками продукта.

Всё вышесказанное приводит к довольно неутешительным выводам о том, что разработчики OPC UA, хотя и стараются сделать свой продукт безопасным, тем не менее, пренебрегают современными практиками и технологиями разработки безопасного программного кода.

По нашей оценке, текущая реализация OPC UA Stack не только не защищает разработчиков от тривиальных ошибок, но и провоцирует их на совершение ошибок, в чём мы убедились на реальных примерах. В условиях современного ландшафта угроз это неприемлемо для продуктов такого широкого применения, как OPC UA. Тем более, это неприемлемо для продуктов, предназначенных для использования в системах промышленной автоматизации.

Центр «Лаборатории Касперского» по реагированию на инциденты информационной безопасности промышленных инфраструктур (Kaspersky Lab ICS CERT) — глобальный проект, нацеленный на координацию действий производителей систем автоматизации, владельцев и операторов промышленных объектов, исследователей информационной безопасности при решении задач защиты промышленных предприятий от кибератак. Усилия Kaspersky Lab ICS CERT направлены в первую очередь на выявление потенциальных и существующих угроз для систем промышленной автоматизации и промышленного интернета вещей.

Kaspersky Lab ICS CERT Ics-cert@kaspersky.com