Статья - Внутренние компоненты Windows IPC: RPC /2/

xss.is/threads/68598

101001100101

Изначально планировалось посвятить LPC и ALPC, но, как оказалось, выкапывать все недокументированные биты и трюки, связанные с этими технологиями, довольно трудоемко. Поэтому я решил сначала опубликовать свои знания о RPC, а затем снова обратиться к ALPC.

Причина, по которой я изначально планировал опубликовать LPC и ALPC перед RPC, заключается в том, что RPC использует ALPC под капотом при локальном использовании и даже больше: RPC - это решение для быстрого локального межпроцессного взаимодействия, поскольку RPC может быть предписано обрабатывать локальное взаимодействие через специальную последовательность протокола ALPC (но вы узнаете это, читая дальше).

В любом случае, урок здесь заключается в том (я думаю), что иногда лучше сделать паузу в каком-то деле, чтобы прояснить ситуацию и продвинуться в чем-то другом, прежде чем вы заблудитесь в чем-то, что просто не готово открыть вам свои тайны.

Введение

Удаленные вызовы процедур (RPC) - это технология, позволяющая осуществлять обмен данными между клиентом и сервером через границы процессов и машин (сетевое взаимодействие). Поэтому RPC является технологией межпроцессного взаимодействия (IPC). Другими технологиями в этой категории являются, например, LPC, ALPC или Named Pipes. Как следует из названия этой категории, RPC используется для осуществления вызовов на удаленные серверы для обмена/передачи данных или для запуска удаленной процедуры. Термин "удаленный" в данном случае не описывает требования к коммуникации. Сервер RPC не обязательно должен находиться на удаленной машине, и теоретически он даже не должен находиться в другом процессе (хотя это имело бы смысл).

Теоретически вы можете реализовать RPC-сервер и клиент в DLL, загрузить их в один и тот же процесс и обмениваться сообщениями, но вы мало что выиграете, так как сообщения все равно будут проходить через другие компоненты вне вашего процесса (например, ядро, но об этом позже), и вы попытаетесь использовать технологию "Inter" Process Communication для "Intra" Process Communication. Более того, сервер RPC не обязательно должен находиться на удаленной машине, он может быть вызван и с локального клиента.

В этой статье вы сможете вместе со мной узнать, что такое RPC, как он работает и функционирует, как реализовать и атаковать RPC-клиенты и серверы.

Эта статья написана с наступательной точки зрения и пытается охватить наиболее важные аспекты поверхности атаки RPC с точки зрения атакующего. Более оборонительный взгляд на RPC можно найти, например, на сайте ДжонатанаДжонсона.

В нижеследующем сообщении будут содержаться некоторые ссылки на код из моих примеров реализации, весь этот код можно найти здесь.

История

Реализация RPC в Microsoft основана на реализации RPC стандарта Distributed Computing Environment (DCE), разработанного Open Software Foundation (OSF) в 1993 году.

"Одной из ключевых компаний, внесших вклад [в реализацию DCE], была Apollo Computer, которая привнесла NCA - 'Network Computing Architecture', ставшую Network Computing System (NCS), а затем и основной частью самого DCE/RPC".
Источник

Microsoft наняла Пола Лича (в 1991 году), одного из инженеров-основателей Apollo, и, возможно, именно так RPC появился в Windows.

Microsoft подстроила модель DCE под свою схему программирования, основала коммуникацию RPC на Named Pipes и представила свою реализацию в Windows 95.

В те времена вы могли задаться вопросом, почему они основали взаимодействие на NamedPipe, ведь Microsoft только что разработала новую технологию под названием Local Procedure Call (LPC) в 1994 году, и кажется, что было бы логично основать технологию под названием Remote Procedure Call на чем-то под названием Local Procedure Call, верно? ... Да, LPC был бы логичным выбором (и я полагаю, что они изначально выбрали LPC), но у LPC был существенный недостаток: он не поддерживал (и до сих пор не поддерживает) асинхронные вызовы (подробнее об этом я расскажу, когда наконец закончу свой пост о LPC/ALPC...), поэтому Microsoft основала его на Named Pipes.

Как мы увидим через некоторое время (раздел RPC Protocol Sequence), при реализации процедур с RPC разработчику необходимо указать библиотеке RPC, какой "протокол" использовать для передачи данных. В оригинальном стандарте DCE/RCP

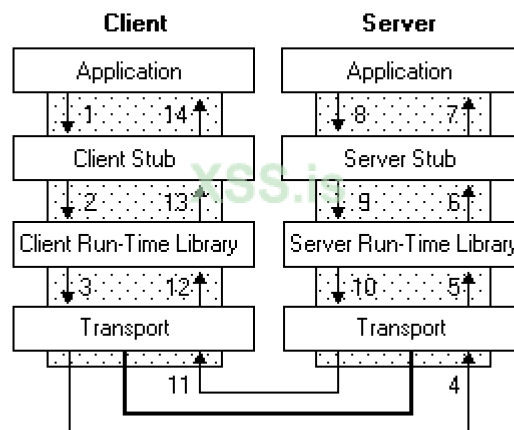
уже были определены 'ncacn_ip_tcp' и 'ncadg_ip_udp' для TCP и UDP соединений. Microsoft добавила 'ncacn_np' для своей реализации на основе Named Pipes (транспортировка через протокол SMB).

Обмен сообщениями RPC

RPC - это технология клиент-сервер с архитектурой обмена сообщениями, похожей на COM (Component Object Model), которая на высоком уровне состоит из следующих трех компонентов:

- Серверный и клиентский процессы, отвечающие за регистрацию интерфейса RPC и связанную с ним информацию о привязке (подробнее об этом позже).
- Серверные и клиентские заглушки, отвечающие за сортировку входящих и исходящих данных.
- Библиотека времени выполнения RPC сервера и клиента (rpcrt4.dll), которая берет данные-заглушки и отправляет их по сети с использованием указанного протокола (примеры и подробности будут следовать ниже)

Визуальный обзор этой архитектуры сообщений можно найти по адресу https://docs.microsoft.com/en-us/windows/win32/rpc/how-rpc-works , как показано ниже:



Позже, в разделе RPC Communication Flow, я представлю обзор шагов от создания RPC-сервера до отправки сообщения, но прежде чем мы сможем погрузиться в это, нам нужно прояснить несколько терминов RPC. Потерпите, пока мы будем копаться во внутренностях RPC. Следующие вещи необходимо знать для того, чтобы освоить RPC.

Если вы запутались в новых терминах и вызовах API, которые вы просто не можете уловить, вы всегда можете перейти к разделу RPC Communication Flow, чтобы получить представление о месте этих вещей в коммуникационной цепочке.

Последовательность протоколов RPC

RPC Protocol Sequence - это постоянная строка, определяющая, какой протокол должна использовать среда выполнения RPC для передачи сообщений. Эта строка определяет, какой протокол RPC, транспортный и сетевой протокол должен использоваться.

Microsoft поддерживает следующие три протокола RPC:

- протокол, ориентированный на соединение Network Computing Architecture (NCACN)
- протокол дейтаграмм архитектуры сетевых вычислений (NCADG)
- локальный удаленный вызов процедур архитектуры сетевых вычислений (NCALRPC).

В большинстве сценариев, где соединение осуществляется по сети, вы будете использовать NCACN, в то время как NCALRPC рекомендуется для локальной связи RPC.

Последовательность протокола - это определенная константная строка, собранная из вышеуказанных частей, например, ncacn_ip_tcp для ориентированного на соединение взаимодействия, основанного на TCP-пакетах.

Полный список констант последовательности протокола RPC можно найти по адресу.

Ниже приведены наиболее важные последовательности протоколов:

ncacn_ip_tcp	Connection-oriented Transmission Control Protocol/Internet Protocol (TCP/IP)
ncacn_http	Connection-oriented TCP/IP using Microsoft Internet Information Server as HTTP proxy
ncacn_np	Connection-oriented named pipes (via SMB.)
ncadg_ip_udp	Datagram (connectionless) User Datagram Protocol/Internet Protocol (UDP/IP)
ncalrpc	Local Procedure Calls (post Windows Vista via ALPC)

Чтобы установить канал связи, среда выполнения RPC должна знать, какие методы (они же "функции") и параметры предлагает ваш сервер и какие данные посылает ваш клиент. Эта информация определяется в так называемом "интерфейсе".

Примечание: Если вы знакомы с интерфейсами в COM, это то же самое.

Чтобы получить представление о том, как можно определить интерфейс, давайте возьмем этот пример из моего примера кода

Interface1.idl
C++:

```
[
    // UUID: A unique identifier that distinguishes this
    // interface from other interfaces.
    uuid(9510b60a-2eac-43fc-8077-aaefbdf3752b),

    // This is version 1.0 of this interface.
    version(1.0),


    // Using an implicit handle here named hImplicitBinding:
    implicit_handle(handle_t hImplicitBinding)

]
interface Example1 // The interface is named Example1
{
    // A function that takes a zero-terminated string.
    int Output(
        [in, string] const char* pszOutput);

    void Shutdown();
}
```

Первое, что необходимо отметить, это то, что интерфейсы определяются в файле Interface Definition Language (IDL). Определения в нем будут позже скомпилированы компилятором Microsoft IDL (midl.exe) в файлы заголовков и исходного кода, которые могут быть использованы сервером и клиентом.

Заголовок интерфейса достаточно самоочевиден с приведенными комментариями - игнорируйте пока инструкцию implicit_handle, мы скоро перейдем к неявным и явным дескрипторам. Тело интерфейса описывает методы, которые этот интерфейс раскрывает, их возвращаемые значения и параметры. Оператор [in, string] в определении параметра функции Output не является обязательным, но помогает понять, для чего используется этот параметр.

Примечание: Вы также можете указать различные атрибуты интерфейса в файле конфигурации приложения (ACF). Некоторые из них, такие как тип связывания (явное или неявное), можно поместить в IDL-файл, но для более сложных интерфейсов вы можете добавить дополнительный ACF-файл для каждого интерфейса.

Привязка RPC

Как только ваш клиент подключается к серверу RPC (позже мы рассмотрим, как это делается), вы создаете то, что Microsoft называет "привязкой". Или, говоря словами Microsoft:

Привязка- это процесс создания логической связи между клиентской программой и серверной программой. Информация, составляющая связь между клиентом и сервером, представлена структурой, называемой дескриптором привязки.

Терминология дескрипторов связывания становится более понятной, когда мы вводим некоторый контекст. Технически существует три типа дескрипторов привязки:

- Неявные
- Явные
- Автоматический

Примечание: Вы можете реализовать пользовательские дескрипторы привязки, как описано здесь, но мы проигнорируем это в данной статье, поскольку это довольно редкое явление, и вы вполне можете обойтись стандартными типами.

Неявные дескрипторы привязки позволяют вашему клиенту подключаться и взаимодействовать с определенным RPC-сервером (указанным UUID в IDL-файле). Недостатком является то, что неявные привязки не являются потокобезопасными, поэтому многопоточные приложения должны использовать явные привязки. Ручки неявного связывания определяются в IDL-файле, как показано в примере IDL-кода выше или в моем примере неявного интерфейса.

Явные дескрипторы привязки позволяют вашему клиенту подключаться и взаимодействовать с несколькими серверами RPC. Явные дескрипторы связывания рекомендуется использовать из-за их потокобезопасности и возможности установки нескольких соединений. Пример определения явного дескриптора привязки можно найти в моем коде здесь.

Автоматическое связывание - это промежуточное решение для ленивого разработчика, который не хочет возиться с дескрипторами связывания и позволяет среде выполнения RPC выяснить, что нужно. Я бы рекомендовал использовать явные дескрипторы, просто чтобы быть в курсе того, что вы делаете.

Зачем мне вообще нужны дескрипторы привязки, спросите вы. Представьте себе дескриптор связывания как представление вашего канала связи между клиентом и сервером, подобно шнуру в телефоне (интересно, сколько людей знают эти

"устройства"...). Учитывая, что у вас есть представление канала связи ("шнур"), вы можете добавить атрибуты к этому каналу связи, например, раскрасить шнур, чтобы сделать его более уникальным.

Подобно этому, ручки привязки позволяют, например, защитить соединение между клиентом и сервером (потому что у вас есть что-то, к чему можно добавить безопасность) и, следовательно, сформировать то, что Microsoft называет "аутентифицированными" привязками.

Анонимные и аутентифицированные привязки

Допустим, у вас работает простой и понятный RPC-сервер, теперь клиент подключается к вашему серверу. Если вы не указали ничего, кроме самого минимума (который я перечислю ниже), такое соединение между клиентом и сервером называется анонимным или неаутентифицированным связыванием, из-за того, что ваш сервер не знает, кто к нему подключился. Чтобы избежать подключения любого клиента и повысить уровень безопасности вашего сервера, вы можете включить три механизма:

1. Вы можете установить флаги регистрации при регистрации интерфейса вашего сервера; И/или
2. Вы можете установить обратный вызов Security с пользовательской процедурой для проверки того, должен ли запрашивающий клиент быть разрешен или запрещен; And/Or
3. Вы можете установить аутентификационную информацию, связанную с вашим дескриптором привязки, чтобы указать поставщика услуг безопасности и SPN для представления вашего RPC-сервера.

Давайте рассмотрим эти три механизма шаг за шагом.

Флаги регистрации

Прежде всего, когда вы создаете свой сервер, вам необходимо зарегистрировать свой интерфейс, например, с помощью вызова RpcServerRegisterIf2 - я покажу вам, где этот вызов вступает в игру в разделе RPC Communication Flow. В качестве четвертого параметра RpcServerRegisterIf2 вы можете указать флаги регистрации интерфейса, например RPC_IF_ALLOW_LOCAL_ONLY, чтобы разрешить только локальные соединения.

Примечание: читайте это как RPC_InterFace_ALLOW_LOCAL_ONLY.

Пример вызова может выглядеть следующим образом:

C++:

```
RPC_STATUS rpcStatus = RpcServerRegisterIf2(
    Example1_v1_0_s_ifspec,         // Interface to register.
    NULL,                           // NULL type UUID
    NULL,                           // Use the MIDL generated entry-point vector.
    RPC_IF_ALLOW_LOCAL_ONLY,        // Only allow local connections
    RPC_C_LISTEN_MAX_CALLS_DEFAULT, // Use default number of concurrent calls.
    (unsigned)-1,                   // Infinite max size of incoming data blocks.
    NULL                            // No security callback.
);
```

Обратные вызовы безопасности

Следующим в списке является обратный вызов безопасности, который вы можете задать в качестве последнего параметра вышеприведенного вызова. Всегда разрешенный обратный вызов может выглядеть следующим образом:

C++:

```
// Naive security callback.
RPC_STATUS CALLBACK SecurityCallback(RPC_IF_HANDLE hInterface, void* pBindingHandle)
{
    return RPC_S_OK; // Always allow anyone.
}
```

Чтобы включить этот обратный вызов Security, просто установите последний параметр функции RpcServerRegisterIf2 в имя вашей функции обратного вызова безопасности, которая в данном случае называется просто "SecurityCallback", как показано ниже:

C++:

```
RPC_STATUS rpcStatus = RpcServerRegisterIf2(
    Example1_v1_0_s_ifspec,         // Interface to register.
    NULL,                           // Use the MIDL generated entry-point vector.
    NULL,                           // Use the MIDL generated entry-point vector.
    RPC_IF_ALLOW_LOCAL_ONLY,        // Only allow local connections
    RPC_C_LISTEN_MAX_CALLS_DEFAULT, // Use default number of concurrent calls.
    (unsigned)-1,                   // Infinite max size of incoming data blocks.
    SecurityCallback                // No security callback.
);
```

Эта функция обратного вызова может быть реализована любым удобным для вас способом, например, вы можете разрешать/запрещать соединения на основе IP-адресов.

Аутентифицированные привязки

Хорошо, пока вы должны знать, что вы можете создавать неявные и явные интерфейсы и использовать несколько вызовов Windows API для настройки вашего RPC-сервера. В предыдущем разделе я добавил, что как только вы зарегистрируете свой сервер, вы можете установить флаги регистрации и (если хотите) также функцию обратного вызова для защиты вашего сервера и фильтрации клиентов, которые могут получить доступ к вашему серверу. Последняя часть головоломки теперь представляет собой дополнительный Windows API, который позволяет серверу и клиенту аутентифицировать вашу привязку (помните, что одно из преимуществ наличия дескриптора привязки заключается в том, что вы можете аутентифицировать свою привязку, например, «покрасить шнур для банки»). Телефон').
… Но зачем/должны ли вы это делать?

Аутентифицированные привязки в сочетании с правильным регистрационным флагом (RPC_IF_ALLOW_SECURE_ONLY) позволяют вашему RPC-серверу гарантировать, что только аутентифицированные пользователи могут подключаться; И, в случае, если клиент разрешает это, позволяет серверу выяснить, кто к нему подключился, выдавая себя за клиента .

Для резервного копирования того, что вы узнали ранее: вы также можете использовать SecurityCallback, чтобы запретить любому анонимному клиенту подключаться, но вам нужно будет реализовать механизм фильтрации самостоятельно, на основе атрибутов, которыми вы управляете.

Пример: вы не сможете определить, например, является ли клиент действительным пользователем домена, потому что у вас нет доступа к этой информации об учетной записи.

Итак, как указать аутентифицированную привязку?

Вы можете аутентифицировать свою привязку на сервере и на стороне клиента. На стороне сервера вы хотите реализовать это, чтобы обеспечить безопасное соединение, а на стороне клиента вам может понадобиться это, чтобы иметь возможность подключаться к вашему серверу (как мы вскоре увидим в матрице доступа)

Аутентификация привязки на стороне сервера: [взято из моего примера кода здесь]

C++:

```
RPC_STATUS rpcStatus = RpcServerRegisterAuthInfo(
    pszSpn,             // Server principal name
    RPC_C_AUTHN_WINNT,    // using NTLM as authentication service provider
    NULL,               // Use default key function, which  is ignored for NTLM SSP
    NULL                // No arg for key function
);
```

Аутентификация привязки на стороне клиента: [Взято из моего примера кода здесь]

C++:

```
RPC_STATUS status = RpcBindingSetAuthInfoEx(
hExplicitBinding,        // the client's binding handle
pszHostSPN,            // the server's service principale name (SPN)
RPC_C_AUTHN_LEVEL_PKT,    // authentication level PKT
RPC_C_AUTHN_WINNT,        // using NTLM as authentication service provider
NULL,            // use current thread credentials
RPC_C_AUTHZ_NAME,        // authorization based on the provided SPN
&secQos            // Quality of Service structure
);
```

Интересным моментом на стороне клиента является то, что вы можете установить структуру **качества обслуживания (QOS)** с помощью аутентифицированного дескриптора привязки. Эта структура QOS может, например, использоваться на стороне клиента для определения уровня **олицетворения** (для получения дополнительной информации см. мою предыдущую публикацию IPC), которую мы позже рассмотрим в разделе Имитация клиента .

Важно отметить :
Установка привязки с проверкой подлинности на стороне сервера не требует проверки подлинности на стороне клиента.

Если, например, на стороне сервера не установлены никакие флаги или установлен только *RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH* , неаутентифицированные клиенты все равно могут подключаться к RPC-серверу.

установка *RPC_IF_ALLOW_SECURE_ONLY* предотвращает привязку клиента без проверки подлинности, поскольку клиент не может установить уровень проверки подлинности (который проверяется с помощью этого флага) без создания привязки с проверкой подлинности.

Известные и динамические конечные точки

И последнее, но не менее важное: мы должны прояснить один последний важный аспект связи RPC: общеизвестные и динамические конечные точки.

Я постараюсь сделать это кратким, так как это также довольно легко понять…

Когда вы запускаете свой сервер RPC, сервер регистрирует интерфейс (как мы уже видели в приведенном выше примере кода с RpcServerRegisterIf2), и ему также необходимо определить, в какой последовательности протоколов (например, 'ncacn_ip_tcp', 'ncacn_np', …) это хочется слушать.

Теперь строки последовательности протокола, которую вы указываете на своем сервере, недостаточно для открытия соединения порта RPC. Представьте, что вы указываете «ncacn_ip_tcp» в качестве последовательности вашего протокола, что означает, что вы указываете своему серверу открыть RPC-соединение, которое принимает соединения через TCP/IP… но… на каком TCP-порту сервер должен фактически открывать соединение?

Подобно *ncacn_ip_tcp* , другим последовательностям протоколов также требуется немного больше информации о том *где* открывать объект соединения:

- ncacn_ip_tcp требует номер порта TCP, например, 9999
- ncacn_np требуется имя именованного канала, например. «\pipe\FRPC-NP»
- ncalrpc требуется имя порта ALPC, например «\RPC Control\FRPC-LRPC»

Предположим на мгновение, что вы указали *ncacn_np* в качестве последовательности протокола и выбрали имя именованного канала «\pipe\FRPC-NP».

Ваш сервер RPC с радостью заработает и теперь ждет подключения клиентов. С другой стороны, клиент должен знать, куда он должен подключаться. Вы сообщаете своему клиенту имя сервера, указываете последовательность протокола как *ncacn_np* и устанавливаете имя именованного канала на то же имя, которое вы определили на своем сервере («\pipe\FRPC-NP»). Клиент успешно подключается, и таким образом вы создали клиент и сервер RPC на основе общеизвестной конечной точки… которая в данном случае: «\pipe\FRPC-NP».

Использование **общеизвестных** конечных точек RPC просто означает, что вы заранее знаете всю информацию о привязке (последовательность протокола и адрес конечной точки) и можете — если хотите — также жестко закодировать эту информацию в своем клиенте и сервере. Использование общеизвестных конечных точек — это самый простой способ создать первое соединение клиент/сервер RPC.

Что такое **динамические конечные точки** и зачем их использовать?

В приведенном выше примере мы выбрали *ncacn_np* и просто выбрали любое произвольное имя именованного канала, чтобы открыть наш сервер, и это сработало просто отлично, потому что мы знали (ну, по крайней мере, мы надеялись), что именованный канал, который мы открыли с этим именем, не уже существует на стороне сервера, потому что мы только что придумали имя. Если теперь мы выберем *ncacn_ip_tcp* в качестве последовательности протоколов, как мы узнаем, какой TCP-порт все еще доступен для нас? Что ж, мы могли бы просто указать, что нашей программе нужен порт 9999 для работы, и предоставить администраторам возможность убедиться, что этот порт не используется, но мы также можем попросить Windows назначить нам свободный порт. И это то, что **динамические конечные точки** . Легко… дело закрыто, пошли пить пиво

Подождите: если нам динамически назначается порт, как клиент узнает, куда подключаться?!…

И это еще одна вещь с динамическими конечными точками: если вы выбрали динамические конечные точки, вам нужен кто-то, чтобы сообщить вашему клиенту, какой порт у вас есть, и этот кто-то является **сопоставления конечных точек RPC** (запущена и работает по умолчанию в вашей системе Windows). Если ваш сервер использует динамические конечные точки ему нужно будет вызвать средство сопоставления конечных точек RPC, чтобы указать ему зарегистрировать свой интерфейс и функции (указанные в файле IDL). Как только клиент попытается создать привязку, он запросит у сервера сопоставления конечных точек RPC совпадающие интерфейсы, и средство сопоставления конечных точек заполнит недостающую информацию (например, TCP-порт) для создания привязки.

Основное преимущество **динамических конечных точек** заключается в автоматическом поиске доступного адреса конечной точки, когда адресное пространство конечной точки ограничено, как в случае с TCP-портами. Именованные каналы и соединения на основе ALPC также можно безопасно выполнять с **общеизвестными** конечными точками, потому что адресное пространство (также известное как произвольное имя канала или порта, которое вы выбрали) достаточно велико, чтобы избежать коллизий.

Мы завершим это фрагментами кода со стороны сервера, чтобы закрепить наше понимание общеизвестных и динамических конечных точек.

Хорошо известная реализация конечной точки
C++:

```
RPC_STATUS rpcStatus;
// Create Binding Information
rpcStatus = RpcServerUseProtseqEp(
    (RPC_WSTR)L"ncacn_np",          // using Named Pipes here
    RPC_C_PROTSEQ_MAX_REQS_DEFAULT, // Ignored for Named Pipes (only used for ncacn_ip_tcp,
but set this anyway)
    (RPC_WSTR)L"\\pipe\\FRPC-NP",   // example Named Pipe name
    NULL                            // No Secuirty Descriptor
);
// Register Interface
rpcStatus = RpcServerRegisterIf2(...) // As shown in the examples above
// OPTIONAL: Register Authentication Information
rpcStatus = RpcServerRegisterAuthInfo(...) // As shown in the example above
// Listen for incoming client connections
rpcStatus = RpcServerListen(
    1,                              // Recommended minimum number of threads.
    RPC_C_LISTEN_MAX_CALLS_DEFAULT, // Recommended maximum number of threads.
    FALSE                           // Start listening now.
);
```

Реализация динамической конечной точки

C++:

```
RPC_STATUS rpcStatus;
RPC_BINDING_VECTOR* pbindingVector = 0;
// Create Binding Information
rpcStatus = RpcServerUseProtseq(
    (RPC_WSTR)L"ncacn_ip_tcp",      // using Named Pipes here
    RPC_C_PROTSEQ_MAX_REQS_DEFAULT, // Backlog queue length for the ncacn_ip_tcp protocol
sequenc
    NULL                            // No Secuirty Descriptor
);
// Register Interface
rpcStatus = RpcServerRegisterIf2(...) // As shown in the examples above
// OPTIONAL: Register Authentication Information
rpcStatus = RpcServerRegisterAuthInfo(...) // As shown in the example above
// Get Binding vectors (dynamically assigend)
rpcStatus = RpcServerInqBindings(&pbindingVector);
// Register with RPC Endpoint Mapper
rpcStatus = RpcEpRegister(
    Example1_v1_0_s_ifspec,             // your interface as defined via IDL
    pbindingVector,                     // your dynamic binding vectors
    0,                                  // We don't want to register the vectors with UUIDs
    (RPC_WSTR)L"MyDyamicEndpointServer" // Annotation used for information purposes only, max
64 characters
);
// Listen for incoming client connections
rpcStatus = RpcServerListen(
    1,                              // Recommended minimum number of threads.
    RPC_C_LISTEN_MAX_CALLS_DEFAULT, // Recommended maximum number of threads.
    FALSE                           // Start listening now.
);
```

Примечание. Если вы используете общеизвестные конечные точки, вы также можете зарегистрировать свой RPC-сервер в локальном сопоставителе конечных точек RPC, вызвав RpcServerInqBindings и RpcEpRegister , если хотите. Вам не нужно делать это, чтобы ваш клиент мог подключиться, но вы могли бы.

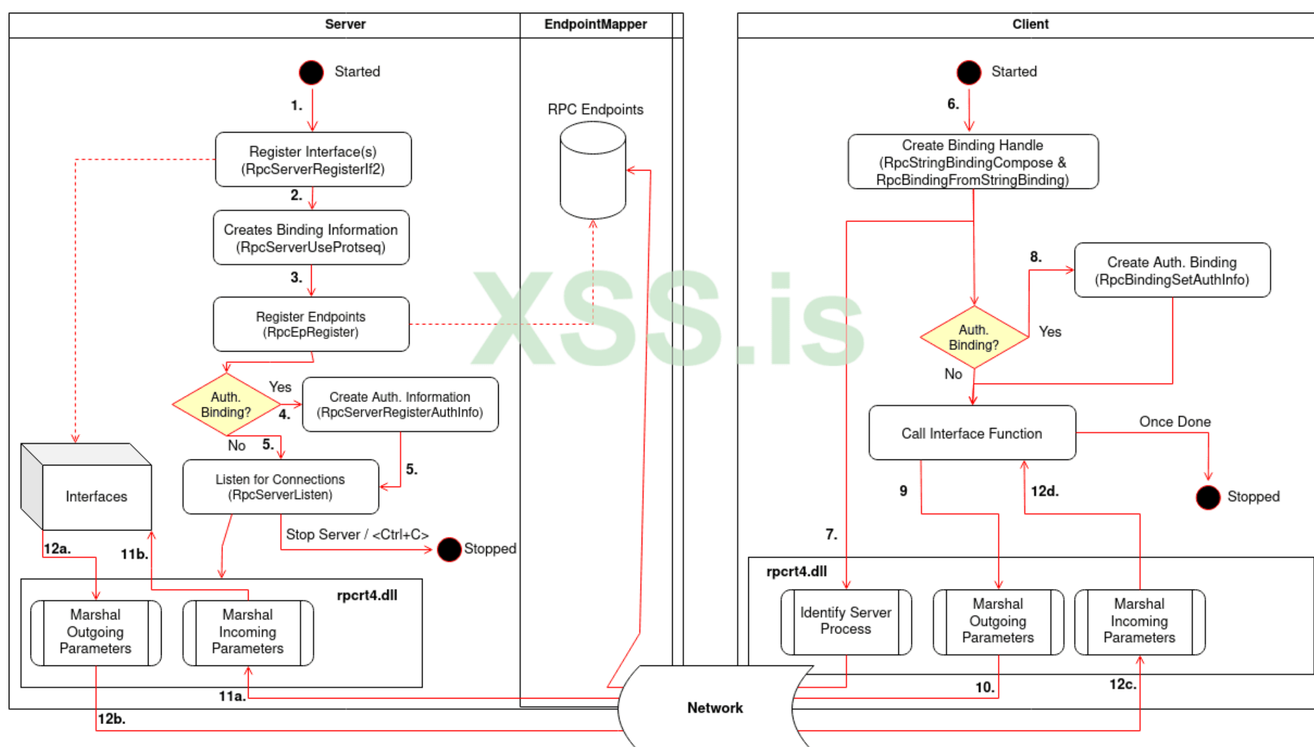
Если вы хотите узнать больше об этом, документацию Microsoft по этой теме можно найти здесь:
https://docs.microsoft.com/en-us/windows/win32/rpc/specifying-endpoints

Коммуникационный поток RPC

Подводя итог всему вышесказанному, можно резюмировать поток общения следующим образом:

1. **Сервер** регистрирует интерфейс(ы), например, используя RpcServerRegisterIf2
2. **Сервер** создает информацию о привязке, используя RpcServerUseProtseq и RpcServerInqBindings (*RpcServerInqBindings* является необязательным для общеизвестных конечных точек)

3. **Сервер** регистрирует конечные точки с помощью RpcEpRegister (необязательно для общеизвестных конечных точек)
4. **Сервер** может зарегистрировать информацию об аутентификации, используя RpcServerRegisterAuthInfo (необязательно)
5. **Сервер** прослушивает клиентские соединения, используя RpcServerListen.
6. **Клиент** создает дескриптор привязки, используя RpcStringBindingCompose и RpcBindingFromStringBinding.
7. **клиента** RPC находит серверный процесс, запрашивая Endpoint Mapper в хост-системе сервера (необходимо только для динамических конечных точек)
8. **Клиент** может аутентифицировать дескриптор привязки с помощью RpcBindingSetAuthInfo (необязательно)
9. **Клиент** выполняет вызов RPC, вызывая одну из функций, определенных в используемом интерфейсе.
10. **Клиентская** библиотека времени выполнения RPC упорядочивает аргументы в недоставке формате
11. Библиотека **сервера** передает маршалированные аргументы в заглушку, которая их демаршалирует, а затем передает подпрограммам сервера.
12. Когда **Сервера** возвращаются, заглушка берет параметры [out] и [in, out] (определенные в IDL-файле интерфейса) и возвращаемое значение, маршалирует их и отправляет маршалированные данные в библиотеку времени выполнения RPC Сервера, который передает их обратно клиенту.



XSS.is

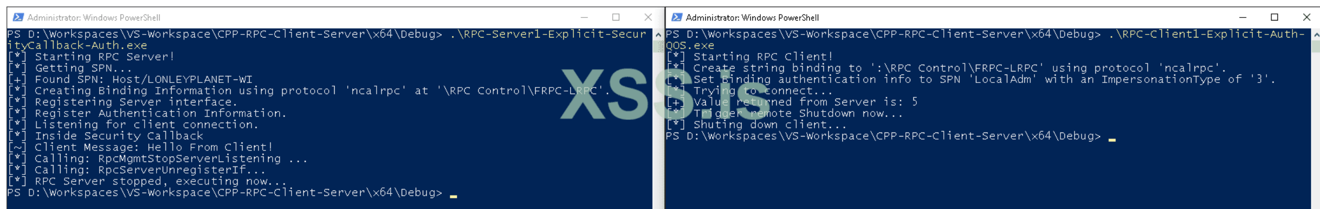
Пример реализации

Как уже упоминалось в начале, приведенные выше примеры взяты из моей пробной реализации, общедоступной по адресу: https://github.com/csandker/InterProcessCommunication-Samples/tree/master/RPC/CPP-RPC-Client-Server .

В этом репозитории вы найдете следующие примеры реализации:

- Базовый неаутентифицированный сервер, поддерживающий неаутентифицированные неявные привязки
- Базовый неаутентифицированный клиент, поддерживающий неаутентифицированные неявные привязки
- Базовый сервер, поддерживающий явные привязки без проверки подлинности
- Базовый сервер, поддерживающий аутентифицированные явные привязки
- Базовый клиент, поддерживающий аутентифицированные явные привязки без QOS
- Базовый клиент, поддерживающий аутентифицированные явные привязки с QOS

Пример того, как выглядят эти PoC, можно увидеть ниже:



XSS.is

Матрица доступа

Хорошо, если вы поняли всю приведенную выше терминологию, вот матрица доступа, которая визуализирует, какой клиент может подключиться к какому серверу.

Примечание. Вы можете подключать неявных клиентов только к неявным серверам, а явных клиентов — к явным серверам. В противном случае вы получите ошибку 1717 (RPC_S_UNKNOWN_IF).

Client\Server		Unauthenticated Binding NoFlags, NoSecurityCallback	Unauthenticated Binding NoFlags, SecurityCallback	Unauthenticated Binding Flags[1], NoSecurityCallback	Unauthenticated Binding Flags[1], SecurityCallback	Authenticated Binding NoFlags, NoSecurityCallback	Authenticated Binding NoFlags, SecurityCallback	Authenticated Binding Flags[1], NoSecurityCallback	Authenticated Binding Flags[1], SecurityCallback
Unauthenticated Binding		Success	Error 5 (Access Denied)	Success	Success	Success	Error 5 (Access Denied)	Success	Success
Authenticated Binding	NoQOS	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Success	Success	Success	Success
Authenticated Binding	QOS	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Error 1747 (RPC_S_UNKNOWN_AUTHN_SERVICE)	Success	Success	Success	Success

XSS.is

QOS Quality of Service
Flags[1] RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH

Поверхность атаки

Наконец… после всех этих разговоров о внутреннем устройстве RPC давайте поговорим о поверхности атаки RPC.

Очевидно, что в цепочке связи RPC могут быть ошибки и 0-day, что всегда сводится к индивидуальному анализу, чтобы понять потенциал эксплойта, но есть также некоторый потенциал эксплуатации общих концепций дизайна RPC, который я выделю ниже.

Дополнительное примечание: если вам известны интересные RPC CVE, пропингуйте меня по адресу / 0xcsandker.

Поиск интересных целей
Итак, прежде чем мы сможем подумать, в какие наступательные игры мы можем играть с RPC, нам нужно сначала найти подходящие цели.

Давайте углубимся в то, как мы можем найти RPC-серверы и клиенты в ваших системах.

RPC-серверы

Напомним, что сервер создается путем указания необходимой информации (последовательность протокола и адрес конечной точки) и вызова API-интерфейсов Windows для создания необходимых внутренних объектов и запуска сервера. Имея это в виду, самый простой способ найти RPC-серверы в вашей локальной системе — это найти программы, которые импортируют эти RPC-API Windows.

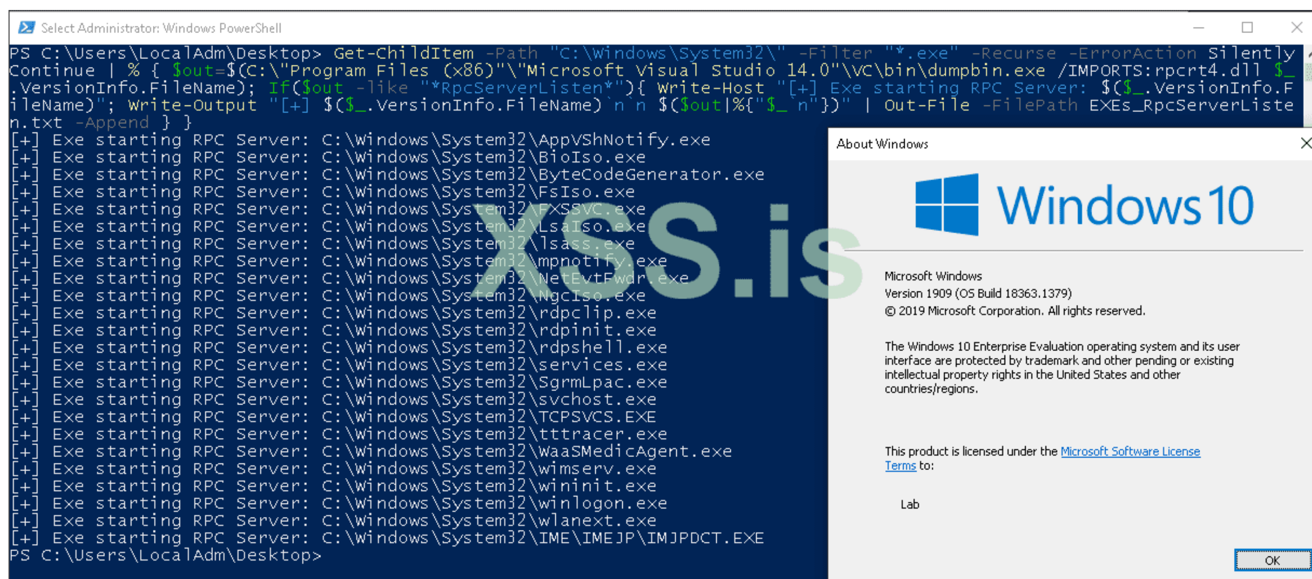
Один из простых способов сделать это — использовать DumpBin , которая в настоящее время поставляется с Visual Studio.

Пример фрагмента кода Powershell для поиска C:\Windows\System32\на недавней Windows10 можно найти ниже:

Code:

```
Get-ChildItem -Path "C:\Windows\System32\" -Filter "*.exe" -Recurse -ErrorAction
SilentlyContinue | % { $out=$(C:\"Program Files (x86)"\"Microsoft Visual Studio
14.0"\VC\bin\dumpbin.exe /IMPORTS:rpcrt4.dll $_.VersionInfo.FileName); If($out -like
"*RpcServerListen*"){ Write-Host "[+] Exe starting RPC Server: $($_.VersionInfo.FileName)";
Write-Output "[+] $($_.VersionInfo.FileName)`n`n $($out|%{"$_`n"})" | Out-File -FilePath
EXEs_RpcServerListen.txt -Append } }
```

Этот фрагмент выводит имена исполняемых файлов на консоль и весь вывод DumpBin в файл *EXEs_RpcServerListen.txt* (чтобы вы могли просмотреть, что на самом деле дает вам DumpBin).



Еще один способ найти интересные RPC-серверы — запросить RPC Endpoint Mapper либо локально, либо в любой удаленной системе. Для этого у Microsoft есть тестовая утилита под названием PortQry (также доступна версия этого инструмента с графическим интерфейсом), которую вы можете использовать следующим образом: C:\PortQryV2\PortQry.exe -n <HostName> -e 135

```
Administrator: Windows PowerShell
PS C:\Users\LocalAdm\Desktop> C:\PortQryV2\PortQry.exe -n GSrv1 -e 135

Querying target system called:

 GSrv1

Attempting to resolve name to IP address...


Name resolved to 10.250.2.120

querying...

TCP port 135 (epmap service): LISTENING

Using ephemeral source port
Querying Endpoint Mapper Database...
Server's response:

UUID: d95afe70-a6d5-4259-822e-2c84da1ddb0d
ncacn_ip_tcp:GSrv1[49664]

UUID: d6b1ad2b-b550-4729-b6c2-1651f58480c3 TEST?
ncacn_np:GSrv1[\\pipe\\FRPC-NP]

UUID: 6b5bdd1e-528c-422c-af8c-a4079be4fe48 Remote Fw APIs
ncacn_ip_tcp:GSrv1[49708]

UUID: 367abb81-9844-35f1-ad32-98f038001003
ncacn_ip_tcp:GSrv1[49694]
```

XSS.is

Этот инструмент предоставляет некоторую информацию об удаленных RPC-интерфейсах, о которых известно программе сопоставления конечных точек (помните, что общеизвестные конечные точки не должны информировать средство сопоставления конечных точек о своих интерфейсах).

Другой вариант — запросить Endpoint Manager напрямую, вызвав RpcMgmtEpEltInqBegin и перебирая интерфейсы через RpcMgmtEpEltInqNext . Пример реализации этого подхода под названием **RPCDump** был включен в потрясающую книгу Криса Макнаба « *Оценка сетевой безопасности* », O'Reilly опубликовал инструмент, написанный на C , здесь (согласно аннотации комментария, кредиты за этот код должны принадлежать Тодду Сабину).

Я перенес этот классный инструмент на VC++ и внес небольшие изменения в удобство использования. Я опубликовал свой форк на https://github.com/csandker/RPCDump .

```
Administrator: Windows PowerShell
PS D:\Workspaces\VS-Workspace\CPP-RPC-Client-Server\x64\Debug> .\RPC-Dump.exe -v GSrv1.SafeAlliance.local
## Testing protseq.: ncacn_ip_tcp

IfId: d95afe70-a6d5-4259-822e-2c84da1ddb0d version 1.0
Annotation:
UUID: 765294ba-60bc-48b8-92e9-89fd77769d91
Binding: ncacn_ip_tcp:GSrv1.SafeAlliance.local[49664]
RpcMgmtInqIfIds succeeded
Interfaces: 4
  76f226c3-ec14-4325-8a99-6a46348418ae v1.0
  894de0c0-0d55-11d3-a322-00c04fa321a1 v1.0
  d95afe70-a6d5-4259-822e-2c84da1ddb0d v1.0
  76f226c3-ec14-4325-8a99-6a46348418af v1.0
RpcMgmtInqServerPrincName failed: 0x6d3
RpcMgmtInqStats succeeded
  Stats[0]: 0
  Stats[1]: 0
  Stats[2]: 0
  Stats[3]: 0

IfId: d6b1ad2b-b550-4729-b6c2-1651f58480c3 version 1.0
Annotation: TEST?
UUID: 00000000-0000-0000-0000-000000000000
Binding: ncacn_np:GSrv1.SafeAlliance.local[\\pipe\\FRPC-NP]
RpcMgmtInqIfIds failed: 0x5
RpcMgmtInqServerPrincName failed: 0x6d3
RpcMgmtInqStats succeeded
  Stats[0]: 0
  Stats[1]: 0
  Stats[2]: 0
  Stats[3]: 0

IfId: 6b5bdd1e-528c-422c-af8c-a4079be4fe48 version 1.0
```

XSS.is

Как показано, этот инструмент также отображает интерфейсы найденных конечных точек RPC вместе с некоторой другой информацией. Я не буду вдаваться в подробности всех этих полей, но если вам интересно, ознакомьтесь с кодом и прочтите документацию по Windows API. Статистика, например, извлекается вызовом RpcMgmtInqStats , где возвращаемые значения упоминаются в Примечания ».

Еще раз помните, что есть только RPC-интерфейсы, зарегистрированные в Endpoint Mapper цели.

RPC-клиенты

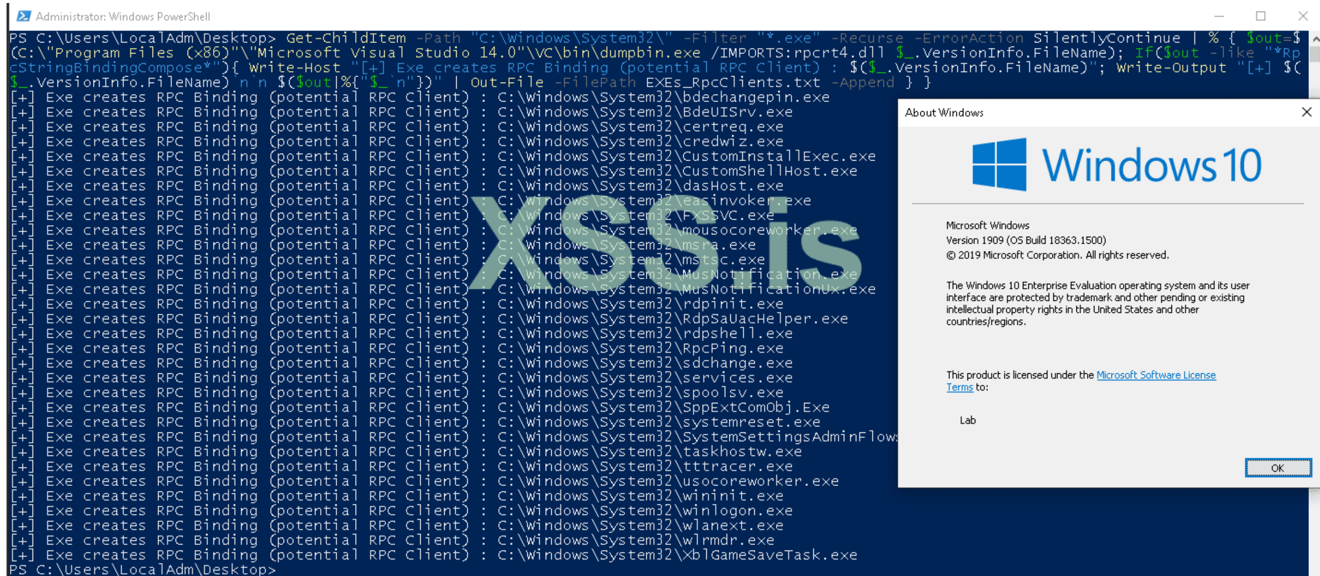
Поиск клиентов, которые подключаются к удаленным или локальным серверам RPC, также может быть интересной целью.
Не существует единого центра, который бы знал, какие клиенты RPC запущены в данный момент, поэтому у вас есть два варианта поиска клиентов:

- Поиск исполняемых файлов/процессов, использующих клиентские RPC API; Или же
- Поймал клиентов на месте

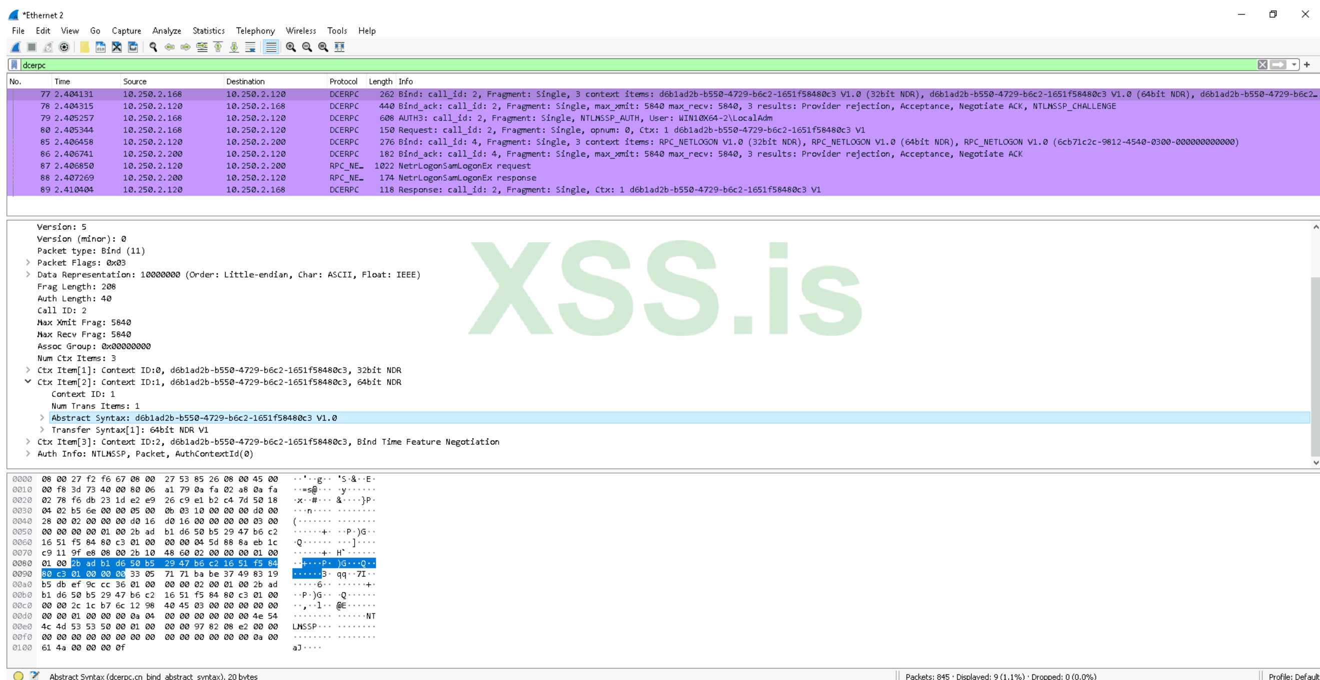
Поиск локальных исполняемых файлов, импортирующих клиентский RPC API, аналогичен тому, что мы уже делали для поиска серверов с помощью DumpBin .
Хорошим Windows API для поиска является RpcStringBindingCompose :

Code:

```
Get-ChildItem -Path "C:\Windows\System32\" -Filter "*.exe" -Recurse -ErrorAction
SilentlyContinue | % { $out=$(C:\"Program Files (x86)"\"Microsoft Visual Studio
14.0"\VC\bin\dumpbin.exe /IMPORTS:rpcrt4.dll $_.VersionInfo.FileName); If($out -like
"*RpcStringBindingCompose*"){ Write-Host "[+] Exe creates RPC Binding (potential RPC Client)
: $($_.VersionInfo.FileName)"; Write-Output "[+] $($_.VersionInfo.FileName)`n`n $($out|%
{"$_`n"})" | Out-File -FilePath EXEs_RpcClients.txt -Append } }
```



Другой способ найти RPC-клиентов — обнаружить их, когда они подключаются к своей цели. Одним из примеров обнаружения клиентов является проверка трафика, передаваемого по сети между двумя системами. В Wireshark есть фильтр DCERPC, который можно использовать для обнаружения подключений.
Пример подключения клиента к серверу показан ниже:



Запрос на привязку — это одна из вещей, которые мы можем искать для идентификации клиентов. В пакете select мы видим клиента, пытающегося привязаться к серверному интерфейсу с UUID «d6b1ad2b-b550-4729-b6c2-1651f58480c3».

Несанкционированный доступ

После того, как вы определили RPC-сервер, который предоставляет интересные функции, которые могут быть полезны для вашей цепочки атак, самое очевидное, что нужно проверить, — это проверить, можете ли вы получить несанкционированный доступ к серверу.

Вы можете либо реализовать свой собственный клиент, например, на основе моего примера реализации , либо обратиться к матрице доступа , чтобы проверить, может ли ваш клиент подключиться к серверу.

Если вы уже глубоко погрузились в обратный инжиниринг RPC-сервера и обнаружили, что сервер устанавливает информацию для проверки подлинности, вызывая RpcServerRegisterAuthInfo со своим SPN и указанным поставщиком услуг, помните, что **привязка сервера с проверкой подлинности не заставляет клиента использовать привязку с проверкой подлинности** . Другими словами: тот факт, что сервер устанавливает аутентификационную информацию, не означает, что клиенту необходимо подключаться через аутентифицированную привязку. Кроме того, при подключении к серверу, который устанавливает информацию для аутентификации, имейте в виду, что **клиентские вызовы с недействительными учетными данными не будут отправлены библиотекой времени выполнения (rpcrt4.dll), однако клиентские вызовы без учетных данных будут отправлены** . Или, говоря словами Microsoft:

> Помните, что по умолчанию безопасность необязательна.
> *Источник: https://docs.microsoft.com/en-us/windows/win32/api/rpcdce/nf-rpcdce-rpcserverregisterifex*

Как только вы подключитесь к серверу, вопрос «что делать дальше?» возникает…

Что ж, теперь вы можете вызывать функции интерфейса, но плохая новость заключается в следующем: сначала вам нужно определить имена и параметры функций, что сводится к обратному проектированию вашего целевого сервера.

Если вам повезло и вы ищете не чистый RPC-сервер, а COM-сервер (COM, особенно DCOM, использует RPC под капотом), сервер может поставляться с библиотекой типов (.tlb), которую вы можете использовать для функции интерфейса поиска.

Я не буду углубляться в библиотеки типов или что-либо еще здесь (сообщение в блоге уже довольно длинное), но моя общая рекомендация для тех, кто находится в такой ситуации, такова: возьмите мой пример кода RPC-клиента и сервера, скомпилируйте его и запустите свой Путешествие по обратному инжинирингу с известным вам образцом кода. В этом конкретном случае позвольте мне добавить еще одну подсказку: в моем примере интерфейса есть функция «Вывод», определенная в файле IDL, эта функция «Вывод» начинается с оператора печати. printf("[~] Client Message: %s\n", pszOutput);, вы можете, например, начать с поиска подстроки [~] Client Messageчтобы выяснить, где находится эта конкретная функция интерфейса.

Имперсонация

Имперсонация клиента также представляет собой интересную поверхность для атак. Я уже немного рассказал о том, что такое имперсонация и как она работает, в предыдущей части серии, если вы пропустили эту статью и вам нужно освежить в памяти информацию об имперсонации, вы найдете объяснения в разделе "Имперсонация" моего последнего сообщения.

Рецепт имперсонификации клиента следующий:

Вам нужен RPC-клиент, подключающийся к вашему серверу.
Клиент должен использовать аутентифицированную привязку (иначе не будет никакой информации безопасности, которую можно было бы выдать за свою)
Клиент не должен устанавливать аутентифицированную привязку Impersonation Level ниже SecurityImpersonation.
... вот и все

Процесс имперсонификации очень прост:

Вызов RpcImpersonateClient из функции интерфейса вашего сервера.

Обратите внимание, что эта функция принимает хэндл привязки в качестве входных данных, поэтому для использования имперсонификации вам нужен сервер с явной привязкой (что вполне логично).

Если этот вызов успешен, контекст потока сервера изменяется на контекст безопасности клиента, и вы можете вызвать GetCurrentThread & OpenThreadToken, чтобы получить токен имперсонации клиента.

Если вы сейчас говорите "WTF изменение контекста безопасности?!", вы найдете ответы в посте IPC Named Pipe
Если вам больше нравится "WTF токен имперсонации?!", вы найдете ответы в моем руководстве по авторизации Windows.

Как только вы вызвали DuplicateTokenEx, чтобы превратить ваш токен Impersonation в первичный токен, вы можете с радостью вернуться в исходный контекст потока сервера, вызвав RpcRevertToSelfEx.

И, наконец, вы можете вызвать CreateProcessWithTokenW, чтобы создать новый процесс с токеном клиента.

Обратите внимание, что это только один из способов создания процесса с токеном клиента, но, на мой взгляд, он достаточно хорошо отображает способ выполнения этих действий, и поэтому я использую этот подход здесь. Пример реализации этого кода можно найти здесь.

Кстати, это та же процедура, которую я использовал для пародирования клиентов Named Pipe в моем предыдущем сообщении.

Как было сказано в рецепте выше, вам просто нужен клиент, который подключается к вашему серверу, и этот клиент должен использовать аутентифицированную привязку.

Если клиент не аутентифицирует свою привязку, то вызов RpcImpersonateClient приведет к ошибке 1764 (RPC_S_BINDING_HAS_NO_AUTH).

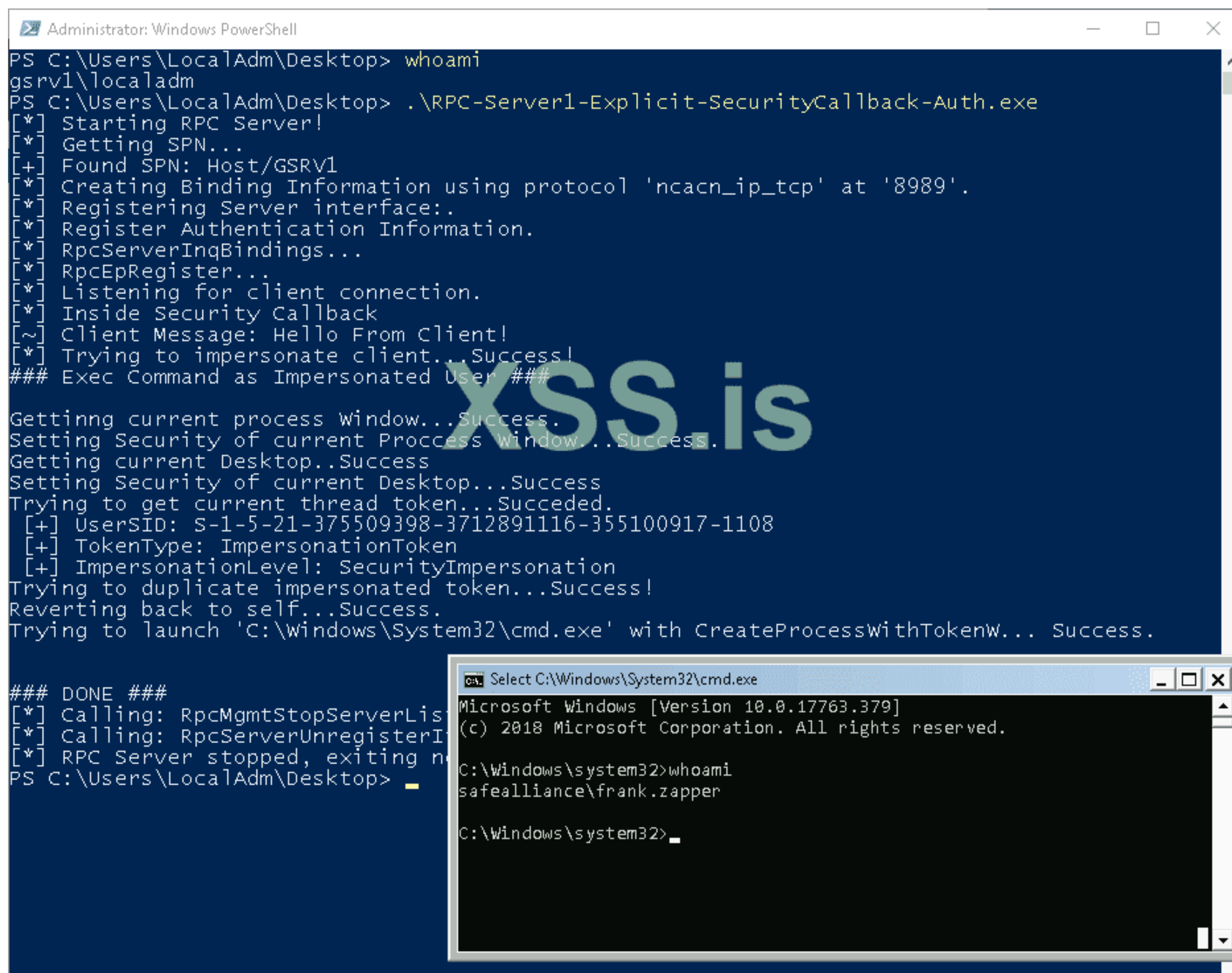
Поиск подходящего клиента, который можно подключить к серверу, сводится к поиску RPC-клиента (см. раздел Поиск RPC-клиентов) и поиску клиента, который можно подключить к серверу. Последнее может оказаться самой сложной частью в этой цепочке эксплойтов, и я не могу дать здесь общих рекомендаций по поиску таких соединений. Одна из причин этого заключается в том, что это зависит от последовательности протоколов, используемых клиентом, где неотвеченный TCP вызов может быть лучше всего обнаружен при прослушивании сети, а неотвеченная попытка соединения Named Pipe может быть также замечена на хост-системе клиента или сервера.

В первой части серии (которая была посвящена Named Pipes) я больше внимания уделил выдаче себя за клиента, поэтому здесь я позволю себе несколько слов. Однако, если вы еще не сделали этого, я бы рекомендовал прочитать об условиях гонки при создании экземпляра, а также об особых вкусах при создании экземпляра. Здесь действуют те же принципы. Более интересным аспектом является то, что я намеренно написал выше: "Клиент не должен устанавливать аутентифицированную привязку уровня имперсонации ниже SecurityImpersonation*"... что звучит как процесс отказа, и именно так оно и есть.

Помните, что вы можете установить структуру Quality of Service (QOS) на стороне клиента при создании аутентифицированной привязки? Как было сказано в разделе Аутентифицированные привязки, вы можете использовать эту структуру для

определения уровня имперсонации при подключении к серверу. Интересно, что если вы не зададите никакой структуры QOS, то по умолчанию будет SecurityImpersonation, что позволяет любому серверу выдавать себя за клиента RPC до тех пор, пока клиент явно не установит уровень обезличивания ниже SecurityImpersonation.

Результат имперсонификации может выглядеть следующим образом:



Неимперсонация сервера

Существует еще одна сторона имперсонализации, которую часто упускают из виду, но которая не менее интересна с точки зрения злоумышленников.

В первой части серии я подробно описал шаги, которые выполняются при выдаче себя за клиента, они в равной степени применимы и к выдаче себя за RPC (и ко всем другим подобным технологиям), где особенно интересны следующие два шага:

Шаг 8: Контекст потока сервера затем изменяется на контекст безопасности клиента.
Шаг 9: Любое действие сервера и любая функция, которую сервер вызывает, находясь в контексте безопасности клиента, выполняются с идентификацией клиента и тем самым выдают себя за него.

Контекст потока сервера изменяется, и все последующие действия выполняются с контекстом безопасности клиента. В приведенном выше разделе (и в моем примере кода) я использовал это для получения токена текущего потока, который затем является токеном клиента, и преобразования его в первичный токен для запуска нового процесса с этим токеном. С тем же успехом я могу просто вызвать любое действие, которое хочу выполнить напрямую, потому что я уже работаю в контексте безопасности клиента. Исходя из названия раздела, вы, возможно, уже догадались, к чему это приведет... что если имперсонификация не удастся, а сервер не проверит это?

Вызов RpcImpersonateClient, функции API, которая делает всю магию имперсонации за вас, возвращает статус операции имперсонации, и для сервера очень важно проверить это.

Если имперсонификация прошла успешно, то после этого вы находитесь в контексте безопасности клиента, но если она не удалась, то вы находитесь в том же старом контексте безопасности, откуда вы вызвали RpcImpersonateClient.
Теперь сервер RPC, вероятно, будет работать от имени другого пользователя (часто также в более высоком контексте безопасности), и в этих случаях он может попытаться выдать себя за своего клиента, чтобы выполнять клиентские операции в более низком, предположительно более безопасном контексте безопасности клиента. Как злоумышленник, вы можете использовать эти случаи для векторов атаки повышения привилегий, заставляя сервер провалить попытку выдачи себя за другого пользователя и тем самым заставляя сервер выполнять клиентские операции в более высоком контексте безопасности сервера.

Рецепт для этого сценария атаки прост:

Вам нужен сервер, который выдает себя за своего клиента и не проверяет тщательно статус возврата RpcImpersonateClient перед выполнением дальнейших действий.

Действия, предпринимаемые сервером после попытки имперсонификации, должны быть уязвимы с точки зрения вашего клиента.

Вам нужно заставить попытку имперсонификации потерпеть неудачу.

Найти локальный сервер, который пытается выдать себя за клиента, - простая задача, если вы прочитали предыдущие разделы и обратили внимание на то, как использовать DumpBin.

Поиск сервера, выполняющего действия в контексте "предполагаемого выдаваемого за клиента", которые могут быть использованы с точки зрения злоумышленников, - это в значительной степени творческий анализ каждого конкретного случая, что делает сервер. Лучший совет для анализа таких случаев - мыслить нестандартно и быть готовым к цепочке из нескольких событий и действий. Довольно простым, но мощным примером может быть файловая операция, выполняемая сервером; возможно, вы можете использовать перекрестки для создания файла в системном пути, защищенном от записи, или заставить сервер открыть именованный канал вместо файла, а затем использовать Named Pipe Impersonation, чтобы выдать себя за сервер...

Последнее в списке - вызвать неудачу попытки имперсонализации сервера, и это самая простая часть работы. Есть два способа добиться этого:

1. Вы можете подключиться с неаутентифицированной привязки; или
2. Вы можете подключиться из аутентифицированной привязки и установить уровень имперсонации в структуре QOS на SecurityAnonymous.

Любое из этих действий приведет к неудачной попытке имперсонации.

Эта техника, кстати, не нова, она широко известна... просто иногда забывается. Возможно, для этой техники есть и более причудливое название, с которым я еще не сталкивался. Microsoft даже специально напоминает об этом в разделе Remarks (они даже дали этому специальный заголовок 'Securtiy Remarks') функции RpcImpersonateClient:

Если вызов RpcImpersonateClient не удается по какой-либо причине, клиентское соединение не обезличивается, и клиентский запрос выполняется в контексте безопасности процесса. Если процесс запущен под высокопривилегированной учетной записью, такой как LocalSystem, или как член административной группы, пользователь может иметь возможность выполнять действия, которые в противном случае были бы запрещены. Поэтому важно всегда проверять возвращаемое значение вызова, и в случае неудачи выдать ошибку; не продолжать выполнение запроса клиента.

NTLM-соединения с аутентификацией MITM

В последних двух разделах рассматривается тот факт, что RPC может использоваться как технология удаленного сетевого взаимодействия и поэтому также имеет интересную поверхность атаки со стороны сети.

Побочное замечание: Я намеренно сформулировал это таким образом; вначале вы могли подумать: "Ну и для чего еще использовать технологию под названием Remote Procedure Call?!". ... Но на самом деле RPC также предназначен для чисто локального использования в качестве обертки для ALPC (я вернусь к этому в третьей части серии, когда разгадаю все тайны ALPC).

В любом случае, если вы используете RPC по проводам и хотите, чтобы ваша привязка была аутентифицирована, вам понадобится сетевой протокол, который будет выполнять аутентификацию за вас. Вот почему второй параметр (AuthnSvc) RpcServerRegisterAuthInfo, который является функцией API, вызываемой на стороне сервера для создания аутентифицированной привязки, позволяет вам определить, какую службу аутентификации вы хотите использовать. Например, вы можете указать Kerberos с постоянным значением RPC_C_AUTHN_GSS_KERBEROS, или вы можете указать RPC_C_AUTHN_DEFAULT, чтобы использовать службу аутентификации по умолчанию, которой, что интересно, является NTLM (RPC_C_AUTHN_WINNT).

Kerberos был установлен в качестве схемы аутентификации по умолчанию с Windows 2000, но RPC все еще использует NTLM по умолчанию.

Поэтому, если вы находитесь в подходящем месте в сети и видите NTLM-соединение, то вы можете сделать с ним две интересные вещи:

Вы можете перехватить хэш ответа на вызов NTLM(v2) и в автономном режиме перебрать пароль пользователя; Или/или
Вы можете перехватить и передать NTLM соединение для получения доступа к другой системе.

Я не хочу глубоко погружаться в эти две темы (если вы дошли до этого места, то наверняка уже достаточно прочитали), поэтому добавлю лишь два замечания:

Форсирование вызова NTLM(v2) очень хорошо известно, поэтому у вас не должно возникнуть проблем с поиском того, как это сделать. В качестве примера посмотрите hashcat mode 5600 на https://hashcat.net/wiki/doku.php?id=example_hashes.
NTLM Relay очень хорошо описана великим Pixis на https://en.hackndo.com/ntlm-relay/. Есть несколько моментов, на которые следует обратить внимание в зависимости от используемого протокола, поэтому обязательно ознакомьтесь с этим постом, если вам интересно.

Соединения GSS_NEGOTIATE с аутентификацией MITM

И последнее, но не менее важное... вы почти дочитали до конца этот пост.

Наряду со схемами сетевой аутентификации на основе NTLM, которые вы получите, если выберете RPC_C_AUTHN_WINNT или RPC_C_AUTHN_DEFAULT в качестве службы аутентификации в вызове RpcServerRegisterAuthInfo, очень часто используемая константа RPC_C_AUTHN_GSS_NEGOTIATE также является интересной целью.

При выборе RPC_C_AUTHN_GSS_NEGOTIATE используется Microsoft's Negotiate SSP для указания клиенту и серверу самостоятельно договориться о том, NTLM или Kerberos следует использовать для аутентификации пользователей. По умолчанию эти переговоры всегда приводят к Kerberos, если клиент и сервер поддерживают его.

Эти переговоры могут быть атакованы с позиции перехватывающей сети, чтобы заставить использовать NTLM вместо Kerberos, эффективно понижая схему аутентификации. Оговорка заключается в том, что для этой атаки требуется подходящая сетевая позиция и отсутствующие подписи. На данный момент я не буду углубляться в эту тему, в основном потому, что я подробно описал процесс и атаку в старой статье здесь: Downgrade SPNEGO Authentication.

Кстати, константы службы аутентификации, упомянутые здесь, можно найти здесь: https://docs.microsoft.com/en-us/windows/win32/rpc/authentication-service-constants.

Вот и все... вы справились!

Ссылки