

Windows Inter Process Communication A Deep Dive Beyond the Surface

 sud0ru.ghost.io/windows-inter-process-communication-a-deep-dive-beyond-the-surface-part-9

Sud0Ru

November 30, 2025



Welcome to the next part of the IPC series, and the final part of the first wave of RPC series. In this post, we will look at the tools you can use to reverse-engineer an RPC server.

This part completes the last two articles where I talked about RPC research tooling. I began with external tools, then moved to internal tools. I split the internal tools into two groups: tools that don't require reverse engineering (covered in the previous [post](#)), and tools that do. Today, we focus on the reverse-engineering side.

Sometimes, during research, you need to go deeper into how an RPC component works, whether it is a client or a server. Tools like RpcView are helpful, but they may not give you everything you need. You might want to inspect the security callback, understand the logic behind each server function, or see which authentication level the client is using.

In this post, I'll share the approaches I use when reversing RPC servers. First, I will start with an automated tool that can perform some basic, high-level reverse engineering for you. After that, we will go deeper using IDA and look at how to reverse both RPC clients and servers.

So let's dive in...

Testbed

For our testbed, we will use the client and server code shown [here](#). We will compile both as 64-bit binaries, which gives us two executables: **server.exe** and **client.exe**. These will be the samples we reverse-engineer throughout this post.

Let's take a quick look at the server code first.

Server Overview

The server listens over **TCP port 41337**:

```
status = RpcServerUseProtseqEp(
    (RPC_CSTR)"ncacn_ip_tcp",
    RPC_C_PROTSEQ_MAX_REQS_DEFAULT,
    (RPC_CSTR)"41337",
    NULL);
```

It uses **NTLM** as its authentication provider:

```
status = RpcServerRegisterAuthInfo(
    pszSpn,
    RPC_C_AUTHN_WINNT, // NTLM
    NULL,
    NULL);
```

The server also registers a **security callback**, and it sets the flags

RPC_IF_ALLOW_SECURE_ONLY | RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH:

```

status = RpcServerRegisterIf2(
    Example_v1_0_s_ifspec,
    NULL,
    NULL,
    RPC_IF_ALLOW_SECURE_ONLY | RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH,
    RPC_C_LISTEN_MAX_CALLS_DEFAULT,
    (unsigned)-1,
    SecurityCallback);

```

The security callback is very simple:

```

RPC_STATUS CALLBACK SecurityCallback(RPC_IF_HANDLE Interface, void* pBindingHandle) {
    printf("The security callback is called");
    return RPC_S_OK;    // Allow any client that connects
}

```

The interface exposes only one RPC function:

```

void PrintString(const char* str) {
    printf("Received string: %s\n", str);
}

```

Client Overview

Now let's look at the client.

The client connects to the server over the network:

```

RpcStringBindingCompose(
    NULL,
    (RPC_CSTR)"ncacn_ip_tcp",
    (RPC_CSTR)"192.168.177.177",
    (RPC_CSTR)"41337",
    NULL,
    &stringBinding);

```

It sets up a **SecurityQOS** structure and uses a high authentication level:

```

RPC_SECURITY_QOS SecurityQOS;
memset(&SecurityQOS, 0, sizeof(SecurityQOS));
SecurityQOS.Version = RPC_C_SECURITY_QOS_VERSION;
SecurityQOS.Capabilities = RPC_C_QOS_CAPABILITIES_DEFAULT;
SecurityQOS.IdentityTracking = RPC_C_QOS_IDENTITY_DYNAMIC;
SecurityQOS.ImpersonationType = RPC_C_IMP_LEVEL_IDENTIFY;

status = RpcBindingSetAuthInfoEx(
    ImplicitHandle,
    NULL,
    RPC_C_AUTHN_LEVEL_PKT,    // packet-level authentication
    RPC_C_AUTHN_WINNT,       // NTLM
    NULL,
    0,
    &SecurityQOS);

```

After setting the authentication info, the client calls the RPC function:

```
PrintString("Hello, RPC Server!");
```

Interface overview:

Finally, here is the interface UUID:

```

[
    uuid(12345678-1234-1234-1234-123456789abc),
    version(1.0),
]

```

PE RPC Scraper

In the beginning I will show you how to use a high-level tool to handle basic reverse-engineering work. The tool we will use is [PE RPC Scraper](#), which is part of Akamai's RPC Toolkit. It's a Python script that analyzes PE files and extracts RPC interface information. You can point it at a single file or let it scan an entire folder.

The script generates a JSON file that contains the RPC interfaces found inside the target DLL or EXE.

When you run the tool without extra options, it performs **static PE parsing**. In this mode, it gives you basic information such as:

- the interface UUID
- the number of functions
- function pointer addresses
- the role (client/server)

The tool also supports external disassemblers such as **IDA Pro** and **Radare2**. When you use the **-d** option, the tool can perform deeper analysis and extract more metadata, such as:

- interface registration sites
- registration flags
- security callback address (if found)
- security descriptor information

Now let's try the tool on our test server and client.

Running the Tool on the Server (No Disassembler)

Command:

```
python pe_rpc_scraper.py server.exe
```

This creates a **rpc_interfaces.json** file with output like this:

```
{
  "server.exe": {
    "12345678-1234-1234-1234-123456789abc": {
      "number_of_functions": 1,
      "functions_pointers": [
        "0x140001040"
      ],
      "function_names": [
        ""
      ],
      "role": "server",
      "flags": "0x40000000",
      "interface_address": "0x140016420"
    }
  }
}
```

From this output, we can see that the tool successfully identified the interface by its UUID. It also detected that the binary exposes one RPC function, and it shows the address of that function and the role of the PE.

Running the Tool with IDA Pro Support

Now let's run it with IDA Pro enabled:

```
python pe_rpc_scraper.py -d idapro server.exe
```

This time, the output includes an additional section called **interface_registration_info**:

```
"interface_registration_info": {
  "0x14000113b": {
    "interface_address": "0x140021120",
    "flags": "0x18",
    "security_callback_addr": "0x0",
    "has_security_descriptor": false,
    "security_callback_info": null,
    "global_caching_enabled": true
  }
}
```

The new information shows the **registration flags**, which are **0x18**. This can be mapped to:

- **0x8** → **RPC_IF_ALLOW_SECURE_ONLY**
- **0x10** → **RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH**

This matches our server's source code.

However, the tool reports **security_callback_addr: 0x0**, which means it didn't detect the security callback correctly. In our case, we know the server *does* have a callback, so this is a limitation of the tool.

Running the Tool on the Client

Now let's check the client. We use the same command as before, but point it at **client.exe**:

```
{
  "client.exe": {
    "12345678-1234-1234-1234-123456789abc": {
      "role": "client",
      "flags": "0x0",
      "interface_address": "0x1400163b0"
    }
  }
}
```

The tool correctly identifies that the executable is an RPC **client** that connects to this UUID.

If we run the tool again with the IDA Pro option, no extra information is added. This is normal because the client does not register an RPC interface, it only consumes one, so there are no registration flags or callbacks to analyze.

IDA: Server Analysis

Now let's load the server into IDA and see how it look.

If you open the server in IDA, the tool will take you straight to **main**, and everything is easy to follow. But in real cases, RPC servers are usually implemented inside DLLs, and they often expose multiple interfaces. Sometimes one DLL contains several servers. Because of that, we need a more systematic way to reverse-engineer RPC servers.




The first thing I like to do is identify the **RPC interface registration functions**, because they reveal the UUIDs of the exposed interfaces.

To start, I search for RPC registration APIs.
If we open the **Imports** tab and type “RPC”, we will see functions like:

- `RpcServerRegisterIf2`
- `RpcServerUseProtseqEpA`
- `RpcServerRegisterAuthInfoA`
- etc.

| Address | Ordinal | Name | Library |
|--|---------|---|---------------------|
|  0000000140016258 | | <code>RpcServerListen</code> | RPCRT4 |
|  0000000140016260 | | <code>NdrServerCall2</code> | <code>RPCRT4</code> |
|  0000000140016268 | | <code>RpcEpRegisterA</code> | RPCRT4 |
|  0000000140016270 | | <code>RpcServerRegisterAuthInfoA</code> | RPCRT4 |
|  0000000140016278 | | <code>RpcMgmtWaitServerListen</code> | RPCRT4 |
|  0000000140016280 | | <code>RpcServerUseProtseqEpA</code> | RPCRT4 |
|  0000000140016288 | | <code>RpcServerRegisterIf2</code> | RPCRT4 |
|  0000000140016290 | | <code>RpcServerInqBindings</code> | RPCRT4 |

If we cross-reference `RpcServerRegisterIf2`, IDA shows us where this function is called inside the server binary.

|  xrefs to <code>RpcServerRegisterIf2</code> | | | |
|--|----|---------|------------------------------|
| Direction | Ty | Address | Text |
|  Up | r | main+9E | call cs:RpcServerRegisterIf2 |
|  Up | p | main+9E | call cs:RpcServerRegisterIf2 |

Line 1 of 2

IDA also shows the actual function arguments as comments, which is very helpful.

```

mov     [rsp+68h+var_28], eax
lea     rax, IfCallbackFn
mov     [rsp+68h+IfCallbackFn], rax ; IfCallbackFn
mov     [rsp+68h+MaxRpcSize], 0FFFFFFFFh ; MaxRpcSize
mov     [rsp+68h+MaxCalls], 4D2h ; MaxCalls
mov     r9d, 18h ; Flags
xor     r8d, r8d ; MgrEvp
xor     edx, edx ; MgrTypeUuid
mov     rcx, cs:IfSpec ; IfSpec
call    cs:RpcServerRegisterIf2

```

Identifying the Interface UUID

Our first task is to identify the interface UUID.

The UUID is located inside the **RPC_SERVER_INTERFACE** structure, which is passed as the **first argument** to **RpcServerRegisterIf2**. IDA usually names this pointer **IfSpec**.

This structure is automatically generated by the MIDL compiler and stored in the server stub. The structure looks like this (from **rpcdcep.h**):

```

typedef struct _RPC_SERVER_INTERFACE {
    unsigned int Length;
    RPC_SYNTAX_IDENTIFIER InterfaceId;
    RPC_SYNTAX_IDENTIFIER TransferSyntax;
    PRPC_DISPATCH_TABLE DispatchTable;
    unsigned int RpcProtseqEndpointCount;
    PRPC_PROTSEQ_ENDPOINT RpcProtseqEndpoint;
    RPC_MGR_EPV* DefaultManagerEvp;
    const void* InterpreterInfo;
    unsigned int Flags;
} RPC_SERVER_INTERFACE, *PRPC_SERVER_INTERFACE;

```

Inside it, the **InterfaceId** member is another structure:

```

typedef struct _RPC_SYNTAX_IDENTIFIER {
    GUID SyntaxGUID;
    RPC_VERSION SyntaxVersion;
} RPC_SYNTAX_IDENTIFIER;

```

The first member is the interface UUID and the second is the version. If we go to the address of **IfSpec** in IDA, we can see the raw bytes that represent this structure.

| | | | |
|-------------------------|---------------|-------------|------------------------------------|
| .rdata:0000000140016420 | unk_140016420 | db 60h ; | ; DATA XREF: .rdata:off_140016480↓ |
| .rdata:0000000140016420 | | | ; .data:IfSpec↓ |
| .rdata:0000000140016421 | | db 0 | |
| .rdata:0000000140016422 | | db 0 | |
| .rdata:0000000140016423 | | db 0 | |
| .rdata:0000000140016424 | | db 78h ; x | |
| .rdata:0000000140016425 | | db 56h ; V | |
| .rdata:0000000140016426 | | db 34h ; 4 | |
| .rdata:0000000140016427 | | db 12h | |
| .rdata:0000000140016428 | | db 34h ; 4 | |
| .rdata:0000000140016429 | | db 12h | |
| .rdata:000000014001642A | | db 34h ; 4 | |
| .rdata:000000014001642B | | db 12h | |
| .rdata:000000014001642C | | db 12h | |
| .rdata:000000014001642D | | db 34h ; 4 | |
| .rdata:000000014001642E | | db 12h | |
| .rdata:000000014001642F | | db 34h ; 4 | |
| .rdata:0000000140016430 | | db 56h ; V | |
| .rdata:0000000140016431 | | db 78h ; x | |
| .rdata:0000000140016432 | | db 9Ah ; S | |
| .rdata:0000000140016433 | | db 0BCh ; % | |
| .rdata:0000000140016434 | | db 1 | |
| .rdata:0000000140016435 | | db 0 | |

You can see:

The first 4 bytes → size of the struct

After that we have RPC_SYNTAX_IDENTIFIER struct:

- The next 16 bytes → the interface UUID
- The next 2 bytes → the version (minor, major)

To make life easier, you can open the **Structures** tab and create an instance of `_RPC_SERVER_INTERFACE` (IDA already has this structure defined). After applying it, IDA will parse everything nicely.

[00000060 BYTES. COLLAPSED STRUCT _RPC_SERVER_INTERFACE. PRESS CTRL-NUMPAD+ TO EXPAND]

| | | | |
|-------------------------|----------------|--|---|
| .rdata:0000000140016420 | stru_140016420 | dd 60h | ; Length |
| .rdata:0000000140016420 | | | ; DATA XREF: .rdata:off_140016480↓ |
| .rdata:0000000140016420 | | | ; .data:IfSpec↓ |
| .rdata:0000000140016420 | | dd 12345678h | ; InterfaceId.SyntaxGUID.Data1 |
| .rdata:0000000140016420 | | dw 1234h | ; InterfaceId.SyntaxGUID.Data2 |
| .rdata:0000000140016420 | | dw 1234h | ; InterfaceId.SyntaxGUID.Data3 |
| .rdata:0000000140016420 | | db 12h, 34h, 12h, 34h, 56h, 78h, 9Ah, 0BCh | ; InterfaceId.SyntaxGUID.Data4 |
| .rdata:0000000140016420 | | dw 1 | ; InterfaceId.SyntaxVersion.MajorVersion |
| .rdata:0000000140016420 | | dw 0 | ; InterfaceId.SyntaxVersion.MinorVersion |
| .rdata:0000000140016420 | | dd 8A885D04h | ; TransferSyntax.SyntaxGUID.Data1 |
| .rdata:0000000140016420 | | dw 1CEBh | ; TransferSyntax.SyntaxGUID.Data2 |
| .rdata:0000000140016420 | | dw 11C9h | ; TransferSyntax.SyntaxGUID.Data3 |
| .rdata:0000000140016420 | | db 9Fh, 0E8h, 8, 0, 2Bh, 10h, 48h, 60h | ; TransferSyntax.SyntaxGUID.Data4 |
| .rdata:0000000140016420 | | dw 2 | ; TransferSyntax.SyntaxVersion.MajorVersion |
| .rdata:0000000140016420 | | dw 0 | ; TransferSyntax.SyntaxVersion.MinorVersion |

The size of this structure is 0x60 bytes (96 decimal).

Now IDA shows the fields clearly, including the interface UUID (GUID).

Checking the Registration Flags and Security Callback

Back in the registration call, we can check the registration flags.

In our case, IDA shows the flags value `0x18`, exactly as the Python script reported.

`0x18` =

- `0x8` → `RPC_IF_ALLOW_SECURE_ONLY`
- `0x10` → `RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH`

This matches the source code.

Now the final thing related to registration is the **security callback**.

The callback pointer appears in the argument list of `RpcServerRegisterIf2` as `IfCallbackFn`.

When we double-click the callback pointer, IDA takes us directly to its function body.

```
IfCallbackFn proc near  
  
arg_0= qword ptr 8  
arg_8= qword ptr 10h  
  
mov     [rsp+arg_8], rdx  
mov     [rsp+arg_0], rcx  
sub     rsp, 28h  
lea     rcx, aTheSecurityCal ; "The security callback is called"  
call    printf  
xor     eax, eax  
add     rsp, 28h  
retn  
IfCallbackFn endp
```

More Useful Functions Around the Server

Near the registration call, we can usually find other important RPC initialization functions. For example:

`RpcServerUseProtseqEpA` → tells us how the server is exposed (the endpoint)

```

sub     rsp, 68h
mov     [rsp+68h+BindingVector], 0
xor     r9d, r9d          ; SecurityDescriptor
lea     r8, Endpoint      ; "41337"
mov     edx, 0Ah          ; MaxCalls
lea     rcx, Protseq       ; "ncacn_ip_tcp"
call    cs:RpcServerUseProtseqEpA

```

`RpcServerRegisterAuthInfoA` → shows the authentication provider (NTLM)

```

mov     [rsp+68h+ServerPrincName], 0
xor     r9d, r9d          ; Arg
xor     r8d, r8d          ; GetKeyFn
mov     edx, 0Ah          ; AuthnSvc
mov     rcx, [rsp+68h+ServerPrincName] ; ServerPrincName
call    cs:RpcServerRegisterAuthInfoA

```

These functions appear close together because the compiler emits the RPC setup in one block.

Finding the Exposed Server Functions (PrintString)

The next important task is finding the functions exposed by the server stub.

These live inside the **MIDL server interpreter structures**, which point to the dispatch table.

The server interface struct `RPC_SERVER_INTERFACE` contains a pointer called `InterpreterInfo`:

```

typedef struct _RPC_SERVER_INTERFACE {
    unsigned int Length;
    RPC_SYNTAX_IDENTIFIER InterfaceId;
    RPC_SYNTAX_IDENTIFIER TransferSyntax;
    PRPC_DISPATCH_TABLE DispatchTable;
    unsigned int RpcProtseqEndpointCount;
    PRPC_PROTSEQ_ENDPOINT RpcProtseqEndpoint;
    RPC_MGR_EPV* DefaultManagerEvp;
    const void* InterpreterInfo; -----> This one
    unsigned int Flags;
} RPC_SERVER_INTERFACE, *PRPC_SERVER_INTERFACE;

```

which points to a `MIDL_SERVER_INFO` structure:

```
typedef struct _MIDL_SERVER_INFO_ {
    PMIDL_STUB_DESC pStubDesc;
    const SERVER_ROUTINE* DispatchTable;
    PFORMAT_STRING ProcString;
    const unsigned short* FmtStringOffset;
    const STUB_THUNK* ThunkTable;
    PRPC_SYNTAX_IDENTIFIER pTransferSyntax;
    ULONG_PTR nCount;
    PMIDL_SYNTAX_INFO pSyntaxInfo;
} MIDL_SERVER_INFO;
```


This structure is also generated by the MIDL compiler.

The most interesting field is:

DispatchTable: pointer to dispatch table that contains pointers to the server functions

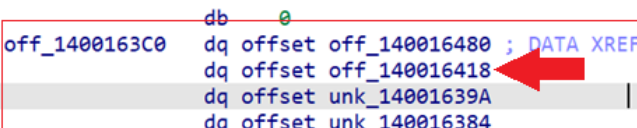
If we go to the **_RPC_SERVER_INTERFACE** struct in IDA, we can find the pointer to **_MIDL_SERVER_INFO**

```
.rdata:0000000140016420 stru_140016420 dd 60h ; Length
.rdata:0000000140016420 ; DATA XREF: .rdata:off_140016480!o
.rdata:0000000140016420 ; .data:IfSpec!o
.rdata:0000000140016420 dd 12345678h ; InterfaceId.SyntaxGUID.Data1
.rdata:0000000140016420 dw 1234h ; InterfaceId.SyntaxGUID.Data2
.rdata:0000000140016420 dw 1234h ; InterfaceId.SyntaxGUID.Data3
.rdata:0000000140016420 db 12h, 34h, 12h, 34h, 56h, 78h, 9Ah, 0BCh; InterfaceId.SyntaxGUID.Data4
.rdata:0000000140016420 dw 1 ; InterfaceId.SyntaxVersion.MajorVersion
.rdata:0000000140016420 dw 0 ; InterfaceId.SyntaxVersion.MinorVersion
.rdata:0000000140016420 dd 8A885D04h ; TransferSyntax.SyntaxGUID.Data1
.rdata:0000000140016420 dw 1CEBh ; TransferSyntax.SyntaxGUID.Data2
.rdata:0000000140016420 dw 11C9h ; TransferSyntax.SyntaxGUID.Data3
.rdata:0000000140016420 db 9Fh, 0E8h, 8, 0, 2Bh, 10h, 48h, 60h; TransferSyntax.SyntaxGUID.Data4
.rdata:0000000140016420 dw 2 ; TransferSyntax.SyntaxVersion.MajorVersion
.rdata:0000000140016420 dw 0 ; TransferSyntax.SyntaxVersion.MinorVersion
.rdata:0000000140016420 db 4 dup(0)
.rdata:0000000140016420 dq offset unk_140016400 ; DispatchTable
.rdata:0000000140016420 dd 0 ; RpcProtseqEndpointCount
.rdata:0000000140016420 db 4 dup(0)
.rdata:0000000140016420 dq 0 ; RpcProtseqEndpoint
.rdata:0000000140016420 dq 0 ; DefaultManagerEpv
.rdata:0000000140016420 dq offset off_1400163C0 ; InterpreterInfo
.rdata:0000000140016420 dd 4000000h ; Flags
```



If we follow that address, IDA shows the pointers inside this struct

```
.rdata:00000001400163BF db 0
.rdata:00000001400163C0 off_1400163C0 dq offset off_140016480 ; DATA XREF: .rdata:stru_140016420!o
.rdata:00000001400163C8 dq offset off_140016418
.rdata:00000001400163D0 dq offset unk_14001639A
.rdata:00000001400163D8 dq offset unk_140016384
.rdata:00000001400163E0 align 40h
.rdata:0000000140016400 unk_140016400 db 1 ; DATA XREF: .rdata:stru_140016420!o
.rdata:0000000140016401 db 0
.rdata:0000000140016402 db 0
.rdata:0000000140016403 db 0
```



The second entry is the **dispatch table pointer**.

If we follow that pointer, we land directly on the address of the RPC-exposed function — in our case, **PrintString**.

```

.rdata:0000000140016417 db 0
.rdata:0000000140016418 off_140016418 dq offset _set_purecall_handler
.rdata:0000000140016418 .DATA XREF: .rdata:0000000140016418

.text:0000000140001040 ; _purecall_handler __cdecl set_purecall_handler(_purecall_handler Handler)
.text:0000000140001040 _set_purecall_handler proc near ; DATA XREF: .rdata:off_140016418lo
.text:0000000140001040 ; .pdata:0000000140023018lo
.text:0000000140001040 arg_0 = qword ptr 8
.text:0000000140001040 mov [rsp+arg_0], rcx
.text:0000000140001040 sub rsp, 28h
.text:0000000140001045 mov rdx, [rsp+28h+arg_0]
.text:0000000140001049 lea rcx, Format ; "Received string: %s\n"
.text:0000000140001055 call printf
.text:000000014000105A add rsp, 28h
.text:000000014000105E retn
.text:000000014000105E _set_purecall_handler endp
.text:000000014000105E ; -----

```

This is how we identify every function implemented by the server stub.

It looks complicated at first, but we will break this down more in the upcoming static analysis part from RPC series.

IDA: Client Analysis

For the client, we usually cannot extract as much information as we can from the server, but there are still some useful things to look at.

Just like before, in the **Imports** tab we search for “RPC”.

A good place to start is the string binding call `RpcStringBindingComposeA`.

```

lea rax, [rsp+88h+String]
mov [rsp+88h+StringBinding], rax ; StringBinding
mov [rsp+88h+Options], 0 ; Options
lea r9, Endpoint ; "41337"
lea r8, NetworkAddr ; "192.168.177.177"
lea rdx, ProtSeq ; "ncacn_ip_tcp"
xor ecx, ecx ; ObjUuid
call cs:RpcStringBindingComposeA

```

IDA automatically annotates the arguments and shows the IP address, port, and protocol sequence.

IDA also identifies the QoS structure and the authentication settings.


```

call    sub_140002100
mov     [rsp+88h+var_38.Version], 1
mov     [rsp+88h+var_38.Capabilities], 0
mov     [rsp+88h+var_38.IdentityTracking], 1
mov     [rsp+88h+var_38.ImpersonationType], 2
lea     rax, [rsp+88h+var_38]
mov     [rsp+88h+SecurityQos], rax ; SecurityQos
mov     dword ptr [rsp+88h+StringBinding], 0 ; AuthzSvc
mov     [rsp+88h+Options], 0 ; AuthIdentity
mov     r9d, 0Ah ; AuthnSvc
mov     r8d, 4 ; AuthnLevel
xor     edx, edx ; ServerPrincName
mov     rcx, cs:Binding ; Binding
call    cs:RpcBindingSetAuthInfoExA

```

The most important part of the client is identifying **which interface UUID it binds to**.

The MIDL compiler generates calls to the RPC runtime, usually one of the `NdrClientCall*` functions.

If we search for `NdrClientCall` in the imports and cross-reference it, IDA takes us to the client stub.

```

.text:0000000140001290      mov     [rsp+arg_0], rcx
.text:0000000140001295      sub     rsp, 28h
.text:0000000140001299      mov     eax, 1
.text:000000014000129E      imul    rax, 0
.text:00000001400012A2      lea     rcx, unk_14001638A
.text:00000001400012A9      add     rcx, rax
.text:00000001400012AC      mov     rax, rcx
.text:00000001400012AF      mov     r8, [rsp+28h+arg_0]
.text:00000001400012B4      mov     rdx, rax ; pFormat
.text:00000001400012B7      lea     rcx, pStubDescriptor ; pStubDescriptor
.text:00000001400012BE      call    NdrClientCall12
.text:00000001400012C3      add     rsp, 28h

```

The function definition is:

```

NdrClientCall12(
    PMIDL_STUB_DESC pStubDescriptor,
    PFORMAT_STRING pFormat,
    ...
);

```

- `pStubDescriptor` → points to the `MIDL_STUB_DESC` structure
- Inside it, the first member is a pointer to an `RPC_CLIENT_INTERFACE` structure which is very similar to `RPC_SERVER_INTERFACE`


```
typedef struct _RPC_CLIENT_INTERFACE {
    unsigned int Length;
    RPC_SYNTAX_IDENTIFIER InterfaceId;    // ← UUID here
    RPC_SYNTAX_IDENTIFIER TransferSyntax;
    PRPC_DISPATCH_TABLE DispatchTable;
    unsigned int RpcProtseqEndpointCount;
    PRPC_PROTSEQ_ENDPOINT RpcProtseqEndpoint;
    ULONG_PTR Reserved;
    const void* InterpreterInfo;
    unsigned int Flags;
} RPC_CLIENT_INTERFACE;
```

Again, the second field (**InterfaceId**) is a struct that contains the UUID and the version:

```
typedef struct _RPC_SYNTAX_IDENTIFIER {
    GUID SyntaxGUID;
    RPC_VERSION SyntaxVersion;
} RPC_SYNTAX_IDENTIFIER;
```

Let's come back to IDA and follow **pStubDescriptor** which is pointer to **MIDL_STUB_DESC** structure and Apply the correct type, IDA resolves everything cleanly.

```
imul    rax, 0
lea     rcx, unk_14001638A
add     rcx, rax
mov     rax, rcx
mov     r8, [rsp+28h+arg_0]
mov     rdx, rax          ; pFormat
lea     rcx, pStubDescriptor ; pStubDescriptor
call    NdrClientCall2
..      -.-
```

```

.rdata:0000000140016410 ; const MIDL_STUB_DESC pStubDescriptor
.rdata:0000000140016410 pStubDescriptor dq offset unk_1400163B0 ; RpcInterfaceInformation
.rdata:0000000140016410 ; DATA XREF: sub_140001290+27f0
.rdata:0000000140016410 dq offset sub_140001000 ; pfnAllocate
.rdata:0000000140016410 dq offset sub_140001020 ; pfnFree
.rdata:0000000140016410 dq offset Binding ; IMPLICIT_HANDLE_INFO.pAutoHandle
.rdata:0000000140016410 dq 0 ; apfnNdrRundownRoutines
.rdata:0000000140016410 dq 0 ; aGenericBindingRoutinePairs
.rdata:0000000140016410 dq 0 ; apfnExprEval
.rdata:0000000140016410 dq 0 ; aXmitQuintuple
.rdata:0000000140016410 dq offset unk_14001637A ; pFormatTypes
.rdata:0000000140016410 dd 1 ; fCheckBounds
.rdata:0000000140016410 dd 50002h ; Version
.rdata:0000000140016410 dq 0 ; pMallocFreeStruct
.rdata:0000000140016410 dd 8010274h ; MIDLVersion
.rdata:0000000140016410 db 4 dup(0)
.rdata:0000000140016410 dq 0 ; CommFaultOffsets
.rdata:0000000140016410 dq 0 ; aUserMarshalQuadruple
.rdata:0000000140016410 dq 0 ; NotifyRoutineTable
.rdata:0000000140016410 dq 1 ; mFlags
.rdata:0000000140016410 dq 0 ; CsRoutineTables
.rdata:0000000140016410 dq 0 ; ProxyServerInfo
.rdata:0000000140016410 dq 0 ; pExprInfo

```

Following the first pointer takes us directly to the struct `RPC_CLIENT_INTERFACE` that contains multiple members and the second is the interface UUID (The first member in `MIDL_SERVER_INFO` struct).

```

.rdata:00000001400163B0 stru_1400163B0 dd 60h ; Length
.rdata:00000001400163B0 ; DATA XREF: .rdata:pStubDescriptor+0
.rdata:00000001400163B0 ; .data:00000001400210D0+0
.rdata:00000001400163B0 dd 12345678h ; InterfaceId.SyntaxGUID.Data1
.rdata:00000001400163B0 dw 1234h ; InterfaceId.SyntaxGUID.Data2
.rdata:00000001400163B0 dw 1234h ; InterfaceId.SyntaxGUID.Data3
.rdata:00000001400163B0 db 12h, 34h, 12h, 34h, 56h, 78h, 9Ah, 0BCh; InterfaceId.SyntaxGUID.Data4
.rdata:00000001400163B0 dw 1 ; InterfaceId.SyntaxVersion.MajorVersion
.rdata:00000001400163B0 dw 0 ; InterfaceId.SyntaxVersion.MinorVersion
.rdata:00000001400163B0 dd 8A885D04h ; TransferSyntax.SyntaxGUID.Data1
.rdata:00000001400163B0 dw 1CE8h ; TransferSyntax.SyntaxGUID.Data2
.rdata:00000001400163B0 dw 11C9h ; TransferSyntax.SyntaxGUID.Data3
.rdata:00000001400163B0 db 9Fh, 0E8h, 8, 0, 2Bh, 10h, 48h, 60h; TransferSyntax.SyntaxGUID.Data4
.rdata:00000001400163B0 dw 2 ; TransferSyntax.SyntaxVersion.MajorVersion
.rdata:00000001400163B0 dw 0 ; TransferSyntax.SyntaxVersion.MinorVersion
.rdata:00000001400163B0 db 4 dup(0)
.rdata:00000001400163B0 dq 0 ; DispatchTable
.rdata:00000001400163B0 dd 0 ; RpcProtseqEndpointCount
.rdata:00000001400163B0 db 4 dup(0)
.rdata:00000001400163B0 dq 0 ; RpcProtseqEndpoint
.rdata:00000001400163B0 dq 0 ; Reserved
.rdata:00000001400163B0 dq 0 ; InterpreterInfo
.rdata:00000001400163B0 dd 0 ; Flags
.rdata:00000001400163B0 db 4 dup(0)
.rdata:0000000140016410 ; const MIDL_STUB_DESC pStubDescriptor

```

This is how we confirm which interface the client is targeting.

I know the IDA part was a bit complicated and full of structures we haven't fully explained yet, but I wanted to show it now because these structures are extremely useful when reversing any RPC server or client. Don't worry, we will revisit all of them in the next parts and go through every detail slowly and clearly.

I hope this final part of the first wave was clear enough, and I hope you enjoyed this introduction to RPC. I wanted to share some of the knowledge I have and present it in a simple way so anyone can start learning this very complex topic.

But we are not done yet — in fact, we have barely started. The next wave will contain a lot more information about RPC, including topics that are not documented anywhere by Microsoft.

So, see you next year in the second wave!