

Windows Inter Process Communication A Deep Dive Beyond the Surface

 sud0ru.ghost.io/windows-inter-process-communication-a-deep-dive-beyond-the-surface-part-7

Sud0Ru

November 9, 2025





Welcome to the new part of the IPC series. This is the sixth part, about RPC, where we will talk about external tools you can use to conduct RPC research.

To get good research results you need a good toolset, tools that help you reach your goal without spending a lot of time building your own. Implementing your own tools can be time-consuming and may need more resources than the research itself.

Unfortunately, there are not many effective tools for RPC because of the complexity and lack of research. Today I will mention the tools I use when I do RPC research. There might be other tools I don't know about, which doesn't mean that tools are ineffective.

As we mentioned before, the client and server can be on different hosts. For this reason the server will be exposed over the network (TCP or named pipes). Because of that, I'll divide the tools into two sets: external and internal. External tools are those you can use from outside the target host, these are the ones I'll cover in today's blog post. Internal tools run inside the host; I will mention them in the next post.

So let's jump in...

External tools

Impacket **rpcdump** script

All external tools I'll mention come from Impacket. Impacket implements RPC in its libraries, so it's easy to build useful scripts on top of it. The first simple tool is **rpcdump**. It lets you dump the endpoint mapper database, which contains the registered interfaces and the endpoints used to access them. Note that an RPC server must register its interface in the endpoint mapper database for it to appear there.

You don't need credentials to dump the database (unless the target Windows host has special policies), because the script performs an unauthenticated bind to the endpoint mapper RPC interface (**epmapper**) and calls the lookup method (OPNUM 2) to dump the whole database.

Below is a photo where we dump the endpoint mapper database on Windows Server 2022.

```

→ ~ rpcdump.py 192.168.177.177
Impacket v0.14.0.dev0+20251107.4500.2f1d6eb2 - Copyright Fortra, LLC and its affiliated companies

[*] Retrieving endpoint list from 192.168.177.177
Protocol: [MS-NRPC]: Netlogon Remote Protocol
Provider: netlogon.dll
UUID : 12345678-1234-ABCD-EF00-01234567CFFB v1.0
Bindings:
    ncacn_ip_tcp:192.168.177.177[49682]
    ncalrpc:[NETLOGON_LRPC]
    ncacn_np:\\DCSRV[\\pipe\\c06894ca3d830bde]
    ncacn_http:192.168.177.177[49674]
    ncalrpc:[NTDS_LPC]
    ncalrpc:[OLEDD74631C610171F1576E36CB72C]
    ncacn_ip_tcp:192.168.177.177[49668]
    ncacn_ip_tcp:192.168.177.177[49664]
    ncalrpc:[MicrosoftLaps_LRPC_0fb2f016-fe45-4a08-a7f9-a467f5e5fa0b]
    ncalrpc:[samss lpc]
    ncalrpc:[SidKey Local End Point]
    ncalrpc:[protected_storage]
    ncalrpc:[lsasspirpc]
    ncalrpc:[lsapolicylookup]
    ncalrpc:[LSA_EAS_ENDPOINT]
    ncalrpc:[lsacap]
    ncalrpc:[LSARPC_ENDPOINT]
    ncalrpc:[securityevent]
    ncalrpc:[audit]
    ncacn_np:\\DCSRV[\\pipe\\lsass]

```

Why this is useful: from the database you can extract the exposed interfaces and full string bindings. You can also get useful details such as which DLL implements the server and which protocol the interface uses which is information that helps you understand the interface functionality.

Impacket **rpcmap** script

rpcmap is more complicated than **rpcdump**. It has many features, let's go through the main ones and how I use them.

```

→ ~ rpcmap.py
Impacket v0.14.0.dev0+20251107.4500.2f1d6eb2 - Copyright Fortra, LLC and its affiliated companies

usage: rpcmap.py [-h] [-brute-uuids] [-brute-opnums] [-brute-versions] [-opnum-max OPNUM_MAX] [-version-max VERSION_MAX]
                [-auth-level AUTH_LEVEL] [-uuid UUID] [-debug] [-ts] [-target-ip ip address] [-port [destination port]]
                [-auth-rpc AUTH_RPC] [-auth-transport AUTH_TRANSPORT] [-hashes-rpc LMHASH:NTHASH] [-hashes-transport LMHASH:NTHASH]
                [-no-pass]
                stringbinding

```

All **rpcmap** commands require a **stringbinding** (that's mandatory). I'll split the explanation into **unauthenticated** and **authenticated** workflows because your goals change depending on whether you have credentials.

A. Unauthenticated binding

Use **-auth-level 1** to force no authentication (auth level 1 = no authentication).

1- List interfaces on an endpoint:

If you run **rpcmap** with a stringbinding (for example, TCP port 135), it will list all interfaces exposed by that endpoint. This works without credentials because **rpcmap** binds to the MGMT interface (which usually allows unauthenticated binds) and calls **rpc_mgmt_inq_if_ids** (OPNUM 0) to retrieve the interfaces as photo below

```
→ ~ rpcmap.py 'ncacn_ip_tcp:192.168.177.177[135]' -auth-level 1
Impacket v0.14.0.dev0+20251107.4500.2f1d6eb2 - Copyright Fortra, LLC and its affiliated companies

Procotol: N/A
Provider: rpcss.dll
UUID: 00000136-0000-0000-C000-000000000046 v0.0

Protocol: [MS-DCOM]: Distributed Component Object Model (DCOM) Remote
Provider: rpcss.dll
UUID: 000001A0-0000-0000-C000-000000000046 v0.0

Procotol: N/A
Provider: rpcss.dll
UUID: 0B0A6584-9E0F-11CF-A3CF-00805F68CB1B v1.1

Procotol: N/A
Provider: rpcss.dll
UUID: 1D55B526-C137-46C5-AB79-638F2A68E869 v1.0

Procotol: N/A
Provider: rpcss.dll
UUID: 412F241E-C12A-11CE-ABFF-0020AF6E7A17 v0.2
```

2- Bruteforce interfaces on an endpoint:

If the target requires authenticated RPC clients (e.g., the Restrict Unauthenticated RPC Clients policy is set to *Authenticated*), the MGMT call will fail. **rpcmap** can still help: it has an internal database of many interface UUIDs and will try to bind each one to see which interfaces exist. This is useful when the normal enumeration call is blocked as photo below

```
→ ~ rpcmap.py 'ncacn_ip_tcp:192.168.177.177[135]' -auth-level 1 -brute-uuids
Impacket v0.14.0.dev0+20251107.4500.2f1d6eb2 - Copyright Fortra, LLC and its affiliated companies

Procotol: N/A
Provider: rpcss.dll
UUID: 00000136-0000-0000-C000-000000000046 v0.0

Protocol: [MS-DCOM]: Distributed Component Object Model (DCOM) Remote
Provider: rpcss.dll
UUID: 000001A0-0000-0000-C000-000000000046 v0.0

Procotol: N/A
Provider: rpcss.dll
UUID: 0B0A6584-9E0F-11CF-A3CF-00805F68CB1B v1.0

Procotol: N/A
Provider: rpcss.dll
UUID: 0B0A6584-9E0F-11CF-A3CF-00805F68CB1B v1.1
```

3- Bruteforce the version of an interface

This functionality is also important for researching, you can check the supported version for a specific interface and then call that version, which could be different from the current one. To do that you can use the command below after you specify the interface UUID and stringbinding.

```
→ ~ rpcmap.py 'ncacn_ip_tcp:192.168.177.177[135]' -brute-versions -uuid AFA8BD80-7D8A-11C9-BEF4-08002B102989 -auth-level 1
Impacket v0.14.0.dev0+20251107.4500.2f1d6eb2 - Copyright Fortra, LLC and its affiliated companies

Protocol: [MS-RPCE]: Remote Management Interface
Provider: rpcrt4.dll
UUID: AFA8BD80-7D8A-11C9-BEF4-08002B102989 v1.0
Versions 0: abstract_syntax_not_supported (version not supported)
Versions 1: success
Versions 2-64: abstract_syntax_not_supported (version not supported)
```

`rpcmap` will try different versions (it defaults to scanning up to version 64; change this with `-version-max`).

Note: if you don't specify a version, `rpcmap` assumes v1.0 by default, this can be a problem if the server only supports v0.0, so always check version ranges when something doesn't bind.

4- Bruteforce opnums for an interface

This is especially important when unauthenticated (where you can check what opnums you can call from interface without authentication). `rpcmap` can try calling each opnum (operation number) with no arguments and report the result. The script iterates opnums (default max 64; change with `-opnum-max`) and records responses such as:

- `rpc_x_bad_stub_data` — wrong arguments were sent
- `nca_s_op_rng_error` — opnum not implemented
- `rpc_s_ok` success — the method ran
- `rpc_access_denied` — needs credentials

A real example: IOXIDResolver. [Airbus Security](#) published a finding showing IOXIDResolver can be used to get network interfaces of a remote Windows host without authentication. When you bruteforce opnums for that interface, opnum **5** returned success with no arguments.

```
→ ~ rpcmap.py 'ncacn_ip_tcp:192.168.177.177[135]' -uuid '99FCFEC4-5260-101B-BBCB-00AA0021347A v0.0' -brute-opnums
Impacket v0.14.0.dev0+20251107.4500.2f1d6eb2 - Copyright Fortra, LLC and its affiliated companies

Protocol: [MS-DCOM]: Distributed Component Object Model (DCOM) Remote
Provider: rpcss.dll
UUID: 99FCFEC4-5260-101B-BBCB-00AA0021347A v0.0
Opnum 0: rpc_x_bad_stub_data
Opnum 1: rpc_x_bad_stub_data
Opnum 2: rpc_x_bad_stub_data
Opnum 3: success
Opnum 4: rpc_x_bad_stub_data
Opnum 5: success
Opnums 6-64: nca_s_op_rng_error (opnum not found)
```

Capturing the response in Wireshark showed the network interface data.


```

SecurityOffset: 25
> StringBinding[1]: TowerId=NCACN_IP_TCP, NetworkAddr="DCSRV"
> StringBinding[2]: TowerId=NCACN_IP_TCP, NetworkAddr="192.168.177.177"
> SecurityBinding[1]: AuthnSvc=0x0009, AuthzSvc=0xffff, PrincName=""
> SecurityBinding[2]: AuthnSvc=0x001e, AuthzSvc=0xffff, PrincName=""
> SecurityBinding[3]: AuthnSvc=0x0010, AuthzSvc=0xffff, PrincName=""
> SecurityBinding[4]: AuthnSvc=0x000a, AuthzSvc=0xffff, PrincName=""
> SecurityBinding[5]: AuthnSvc=0x0016, AuthzSvc=0xffff, PrincName=""
> SecurityBinding[6]: AuthnSvc=0x001f, AuthzSvc=0xffff, PrincName=""
> SecurityBinding[7]: AuthnSvc=0x000e, AuthzSvc=0xffff, PrincName=""

```

When `rpcmap` runs into issues

In some research situations, when you are not testing against a default system, you may face some problems with `rpcmap`.

Let's assume you are searching for interfaces that can bind without authentication when the **Restrict Unauthenticated RPC Clients** policy is set to **"Authenticated"**.

If the host has the *Restrict Unauthenticated RPC Clients* policy set to *Authenticated*, `rpcmap` will often immediately get `rpc_s_access_denied`. This happens because `rpcmap` first calls the MGMT interface on the endpoint to check access, and that MGMT call may itself require authentication.

Quick workaround: edit the `rpcmap` script to skip the initial MGMT check. Comment out the two lines in the `do()` function where it calls the MGMT interface (lines 125 and 128).

```

121
122 def do(self):
123     try:
124         # Connecting to MGMT interface
125         # self.__dce.bind(mgmt.MSRPC_UUID_MGMT)
126
127         # Retrieving interfaces UUIDs from the MGMT interface
128         # ifids = mgmt.hinq_if_ids(self.__dce)
129
130         # If -brute-uuids is set, bruteforcing UUIDs instead of parsing ifids
131         # We must do it after mgmt.hinq_if_ids to prevent a specified account from being locked out
132         if self.__brute_uuids:
133             self.bruteforce_uuids()
134         return
135

```

After that change, `rpcmap` will attempt the direct bind and call bruteforces without relying on the MGMT check. This can reveal interfaces that are registered with `RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH` (i.e., allowed without auth) and this is how I discovered the [enumerating domain users without authentication](#)

B. Authenticated Binding:

Now let's assume you want to search for RPC interfaces using authenticated binding. `rpcmap` provides several parameters related to authentication.

The first and most important thing is setting the authentication level you want to use. This is useful for checking what levels the RPC server supports. You can do that using the `-auth-level` option.

If you set the authentication level to more than 1, you will need to provide credentials for authentication. `rpcmap` allows this using the `-auth-rpc` parameter, followed by the username and password (for example, `username:password`).

You can go even further and specify authentication for the transport layer you are using. For example, if you are using named pipes, `rpcmap` allows you to provide SMB authentication using the `-auth-transport` option.

In addition, you can use NT hashes for authentication by using `-hashes-rpc` and `-hashes-transport`.

RPC clients using a NULL session

One important thing you may face in your research is accessing an RPC interface over SMB while using a NULL session at the SMB level.

The problem is that this cannot be done with the Windows API. When you use the Windows API to talk to an RPC server, you can choose “no authentication” at the RPC level. But if the endpoint is a named pipe (SMB transport), the Windows API will always use your current Windows login to authenticate to SMB. In other words, you cannot force an SMB-level null session through the Windows API — the OS will always supply your session credentials for SMB even if the RPC layer is set to no authentication.

Below is a simple Impacket-based client example that shows how to open a named-pipe transport with empty credentials (a null session) and bind to an interface.

The client is very simple. First, we specify the username, password, and the IP address of the target. Since we’re trying to access the named pipe through a null session, we will use empty credentials

```
domain = ''
username = ''
password = ''
remoteIP = '192.168.177.177'
```

Next, we create the **string binding** to access the RPC server by specifying our target named pipe name. The string binding tells Impacket to use SMB as the transport layer

```
stringbinding = r'ncacn_np:%s[\pipe\MyPipe]' % remoteIP
```

Then we create the transport and set the credentials. Even though we’re performing a null session (so empty credentials suffice), I’ve included `set_credentials()` in case you want to test other scenarios

```
rpctransport = transport.DCERPCTransportFactory(stringbinding)
rpctransport.set_credentials(username, password)
```


After that, we obtain the DCE/RPC connection object and set the authentication level to 1 (which corresponds to `RPC_C_AUTHN_LEVEL_NONE`).

```
dce = rpctransport.get_dce_rpc()
dce.set_auth_level(1)
```

Next, we bind to our interface, which is already defined as a global variable in your code (e.g., `MSRPC_UUID_EXAMPLE`)

```
dce.connect()
dce.bind(MSRPC_UUID_EXAMPLE)
```

Finally, we call the RPC function with opnum number 0, and receive the response

```
dce.call(0, b'')
dce.recv()
```

Calling Custom RPC Functions with Impacket

Final thing I want to tell you about external tools.

Say you want to call a function over an interface remotely using Impacket, but that function or interface is not already implemented in Impacket. Impacket gives you a large library of Windows types and NDR helpers so you can build your own function call. After you determine the function arguments and their types, you can define the call using Impacket's NDR classes and call it with the right arguments.

As an example, I used a function for enumerating domain users. It is already implemented in Impacket under `impacket/dcerpc/v5/nrpc.py`. The function takes seven arguments.

```
NET_API_STATUS DsrGetDcNameEx2(
    [in, unique, string] LOGONSRV_HANDLE ComputerName,
    [in, unique, string, range(0,256)] wchar_t* AccountName,
    [in] ULONG AllowableAccountControlBits,
    [in, unique, string, range(0,256)] wchar_t* DomainName,
    [in, unique] GUID* DomainGuid,
    [in, unique, string, range(0,256)] wchar_t* SiteName,
    [in] ULONG Flags,
    [out] PDOMAIN_CONTROLLER_INFO* DomainControllerInfo
);
```

In Impacket, the function is defined as a class that inherits from `NDRCALL`. `NDRCALL` represents an NDR-encoded RPC request. Inside the class they set the `opnum` for the RPC function. Then they define a tuple-of-tuples where each inner tuple has two elements:

1. The field name (a string)
2. The field type (an NDR type/class, e.g. `LPWSTR`, `ULONG`, etc.)

```

# 3.5.4.3.1 DsrGetDcNameEx2 (Opnum 34)
class DsrGetDcNameEx2(NDRCALL):
    opnum = 34
    structure = (
        ('ComputerName', PLOGONSRV_HANDLE),
        ('AccountName', LPWSTR),
        ('AllowableAccountControlBits', ULONG),
        ('DomainName', LPWSTR),
        ('DomainGuid', PGUID),
        ('SiteName', LPWSTR),
        ('Flags', ULONG),
    )

```

To call this function you need a binding handle (like the one we built in the null-session example). Instead of using `dce.call()`, you construct a request object, fill its fields, and send it with `dce.request()`.

Example:

```

request = DsrGetDcNameEx2()
request['ComputerName'] = "DCSRV"
request['AccountName'] = checkNullString(accountName)
request['AllowableAccountControlBits'] = allowableAccountControlBits
request['DomainName'] = checkNullString(domainName)
request['DomainGuid'] = domainGuid
request['SiteName'] = checkNullString(siteName)
request['Flags'] = flags

response = dce.request(request)

```

Notes:

- `checkNullString()` is a helper used in many Impacket modules to convert empty Python strings into the correct NDR representation for NULL or empty strings.
- Building the request this way gives you full control of the arguments and their NDR types, so you can implement and call functions that Impacket doesn't provide out of the box.

Hope that was useful. I think I covered everything you need to conduct RPC external research. Let's meet in the next part, where I will talk about some interesting internal tools.