

Windows Inter Process Communication A Deep Dive Beyond the Surface

 sud0ru.ghost.io/windows-inter-process-communication-a-deep-dive-beyond-the-surface-part-4

Sud0Ru

June 21, 2025



Welcome to the fourth part of the IPC series — and the third part focused specifically on RPC. Today, we're going to talk about: **RPC security**, which will help complete the picture we've been building around how RPC works under the hood.

RPC security is a **deep and complex** topic, mostly because there are so many different ways to secure RPC servers — and those methods have evolved significantly over the years. Today, and in the next parts, I'll try to cover all the major aspects of RPC security so you get a clear understanding of the options, mechanisms, and challenges involved.

Today, we're going to dig into **binding authentication** — how your server can register an authentication service to allow clients to authenticate, how the client performs that authentication, and how the client can specify the **authentication level**. We'll also look at what each level means in practice and how it shows up in network traffic.

As with the previous parts, I want to start by mentioning the resources behind this work. This post is based on my own research, along with:

- Microsoft's official documentation (MSDN),
- The excellent work by @0xcsandker on [offensive Windows IPC](#),
- [James Forshaw's blog post](#),
- And [Ben Barnea's detailed write-up on the Akamai blog](#).

This will likely be the most challenging parts of the series so far — so get ready, and let's dive in!

RPC Security

The Remote Procedure Call (RPC) runtime provides a standardized interface for authentication — available to both clients and servers. On the server side, it uses the system's authentication services (like NTLM or Kerberos) to validate connections. Applications can use **authenticated RPC** to ensure that:

- All incoming calls are from trusted clients, and
- All responses come from verified servers.

There are several ways to secure an RPC server, but before we get into the different methods, let's start with the basics: **authentication in RPC**.

Binding Authentication

An RPC server can register supported authentication methods, such as **Kerberos** or **NTLM**. The client is then expected to perform **binding authentication** using one of the supported methods.

The client can also specify:

- The authentication **protocol** it wants to use
- The **authentication level** (which defines how much protection the call has)

Registering Authentication: `RpcServerRegisterAuthInfo`

To enable authentication in your RPC server, you need to register an authentication service using the `RpcServerRegisterAuthInfo` function.

Here's the function definition:

```
RPC_STATUS RpcServerRegisterAuthInfo(  
    RPC_CSTR                ServerPrincName,  
    unsigned long            AuthnSvc,  
    RPC_AUTH_KEY_RETRIEVAL_FN GetKeyFn,  
    void                     *Arg  
);
```

Let's check the arguments:

1. **ServerPrincName**

This is the **server principal name**, and it must match the authentication service you're using.

- For **Kerberos**, this is typically an **SPN** (Service Principal Name), and it needs to be registered correctly in the Kerberos database for authentication to succeed.
- For **NTLM**, this parameter can be **NULL**.

2. **AuthnSvc**

This defines which **authentication service** to use. Common options include:

- **RPC_C_AUTHN_WINNT** – NTLM
- **RPC_C_AUTHN_GSS_KERBEROS** – Kerberos

Here's the full list of supported values:

```
#define RPC_C_AUTHN_NONE           0
#define RPC_C_AUTHN_DCE_PRIVATE   1
#define RPC_C_AUTHN_DCE_PUBLIC    2
#define RPC_C_AUTHN_DEC_PUBLIC    4
#define RPC_C_AUTHN_GSS_NEGOTIATE 9
#define RPC_C_AUTHN_WINNT         10
#define RPC_C_AUTHN_GSS_SCHANNEL  14
#define RPC_C_AUTHN_GSS_KERBEROS  16
#define RPC_C_AUTHN_DPA           17
#define RPC_C_AUTHN_MSN           18
#define RPC_C_AUTHN_KERNEL        20
#define RPC_C_AUTHN_DIGEST        21
#define RPC_C_AUTHN_NEGO_EXTENDER 30
#define RPC_C_AUTHN_PKU2U         31
#define RPC_C_AUTHN_LIVE_SSP       32
#define RPC_C_AUTHN_LIVEXP_SSP     35
#define RPC_C_AUTHN_CLOUD_AP       36
#define RPC_C_AUTHN_MSONLINE      82
```

3. **GetKeyFn**

A custom key-retrieval function (usually **NULL** for NTLM or Kerberos).

It's only required for custom authentication providers.

4. **Arg**

An optional argument for the key-retrieval function. Again, for NTLM, this is typically **NULL**.

In our work, we'll focus on **NTLM** because it's simpler to configure.

Here's how registering NTLM authentication might look in code:

```
status = RpcServerRegisterAuthInfo(
    NULL,          // Server principal name (NULL for NTLM)
    RPC_C_AUTHN_WINNT, // NTLM as authentication service
    NULL,          // Default key function (ignored for NTLM)
    NULL           // No argument for key function
);
```

This tells the RPC runtime to accept and use NTLM authentication from clients.

Client-Side Binding Authentication:

After seeing how the **server registers its authentication service**, let's move to the **client side** and see how the client actually uses that authentication — this process is known as **binding authentication**.

As discussed earlier, a **binding handle** is just a pointer to a structure used by the RPC runtime to store connection-related data — and **authentication information is one part of that structure**. When the client configures authentication for the handle, that's called **binding authentication**.

To configure this, the client uses the API:

```
RPC_STATUS RpcBindingSetAuthInfoExA(
    RPC_BINDING_HANDLE      Binding,
    RPC_CSTR               ServerPrincName,
    unsigned long           AuthnLevel,
    unsigned long           AuthnSvc,
    RPC_AUTH_IDENTITY_HANDLE AuthIdentity,
    unsigned long           AuthzSvc,
    RPC_SECURITY_QOS        *SecurityQos
);
```

Let's break the arguments:

1. **Binding**
The binding handle (already created via `RpcBindingFromStringBinding`).
2. **ServerPrincName**
The Server Principal Name (SPN). This must match the expected name for **Kerberos**. If you're using **NTLM**, this can be `NULL`.
3. **AuthnLevel**
The **authentication level** — this defines how secure you want the connection to be (e.g., none, connect-only, per-call, encrypted). We'll break this down in a minute.
4. **AuthnSvc**
The **authentication service** to use. For example:
 - `RPC_C_AUTHN_WINNT` for NTLM
 - `RPC_C_AUTHN_GSS_KERBEROS` for Kerberos

This must match the service that the server registered which we saw earlier

`RpcServerRegisterAuthInfo`.

5. `AuthIdentity`

A pointer to the **identity credentials** (username, domain, and password). When using **NTLM**, you use the `SEC_WINNT_AUTH_IDENTITY_A` structure:

```
typedef struct _SEC_WINNT_AUTH_IDENTITY_A {
    unsigned char *User;
    unsigned long UserLength;
    unsigned char *Domain;
    unsigned long DomainLength;
    unsigned char *Password;
    unsigned long PasswordLength;
    unsigned long Flags;
} SEC_WINNT_AUTH_IDENTITY_A, *PSEC_WINNT_AUTH_IDENTITY_A;
```

Example in code:

```
SEC_WINNT_AUTH_IDENTITY identity;
memset(&identity, 0, sizeof(identity));

identity.User = (unsigned short*)L"MyUserName";
identity.UserLength = wcslen((wchar_t*)identity.User);

identity.Domain = (unsigned short*)L"MyDomain"; // or empty for local
identity.DomainLength = wcslen((wchar_t*)identity.Domain);

identity.Password = (unsigned short*)L"MyPassword";
identity.PasswordLength = wcslen((wchar_t*)identity.Password);

identity.Flags = SEC_WINNT_AUTH_IDENTITY_UNICODE;
```

6. `AuthzSvc`

Defines the **authorization service**. Often set to `RPC_C_AUTHZ_NONE`.

7. `SecurityQos`

A pointer to the `RPC_SECURITY_QOS` structure, used mainly to control **impersonation**. This tells the server **how much it can act on behalf of the client**.

(We'll cover impersonation in IPC in a later part. For now, this can be `NULL`.)

Let's now look at the `AuthnLevel` parameter — this controls how secure the communication should be. The client sets this, and it affects whether authentication is required, and how much protection is applied to the RPC traffic.

There are six levels of authentication as you can see in the table below

RPC Authentication Levels

Level	Name	Description
0	RPC_C_AUTHN_LEVEL_DEFAULT	Uses the default authentication level for the security provider. Often maps to Connect or None , depending on context.
1	RPC_C_AUTHN_LEVEL_NONE	No authentication. The client is not identified .
2	RPC_C_AUTHN_LEVEL_CONNECT	Authenticates only at the time of connection . Once connected, no further identity checks.
3	RPC_C_AUTHN_LEVEL_CALL	Authenticates each individual call to ensure it's coming from the authenticated user.
4	RPC_C_AUTHN_LEVEL_PKT	Ensures data integrity – the data hasn't been tampered with during transit.
5	RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	Same as level 4. Used synonymously.
6	RPC_C_AUTHN_LEVEL_PKT_PRIVACY	Ensures data confidentiality by encrypting the data (in addition to integrity). This is the most secure level.

Worth to note that The server has no built-in way to enforce a **minimum authentication level** during binding. That means it's possible for a client to connect with a lower level than you expect.

To truly enforce secure authentication, the server must implement additional validation logic — such as using the `RpcBindingInqAuthClient` API to check what authentication level the client used, and grant or deny access accordingly. Another option is to use specific flags to prevent unauthenticated access.

We'll cover both approaches in more detail when we will talk about other security measures.

Now let's break down what **authentication levels** actually mean in RPC — and how they appear in network traffic.

Let's explore some of them:

1. Authentication Level: None

As the name suggests, this level means **no authentication** is used. The client doesn't send any credentials, and the server has no way to verify the identity of the client.

This level is equal to not using authentication at all, so we will just use our old client/server but the communication this time will be over the network, you can check the client and server [here](#).

Note: We'll go deeper into RPC over the network when we talk about marshalling later. For now, this is just to demonstrate how authentication looks in traffic.

I used a basic client and server setup with network communication.

To get the same results as mine, you should run the **server and client on different machines**, and don't forget to **update the IP address** of the RPC server in the client's binding handle accordingly.

Here's what you should do:

1. Start **Wireshark** and begin capturing network traffic.
2. Run the **server**.
3. Then run the **client**.

120	37.498646	192.168.177.101	192.168.177.100	DCERPC	170 Bind: call_id: 2, Fragment: Single, 2 context items: 12345678-1234-1234-1234-123456789abc V1.0 (32bit NDR), 12345678-1234-1234-1234-123456789abc V1.0
121	37.524443	192.168.177.100	192.168.177.101	DCERPC	138 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2 results: Acceptance, Negotiate ACK
122	37.525897	192.168.177.101	192.168.177.100	DCERPC	109 Request: call_id: 2, Fragment: Single, opnum: 0, Ctx: 0 12345678-1234-1234-1234-123456789abc V1
123	37.535857	192.168.177.100	192.168.177.101	DCERPC	82 Response: call_id: 2, Fragment: Single, Ctx: 0 12345678-1234-1234-1234-123456789abc V1

In the captured traffic above, you'll see something like this:

1. The **first packet** is a *bind request* — the client asking to bind to the interface.
2. The **second packet** is a *bind response* from the server.
3. The **third packet** is the actual remote procedure call (in our case, **PrintString**).
4. The **final packet** is the response returned by the server.

Now, if you open the RPC bind request and check the headers, you'll see that the **authentication length is zero**. This confirms that the authentication level used is: **RPC_C_AUTHN_LEVEL_NONE**

In other words — no authentication was used at all.

Here's a screenshot that shows this behavior in Wireshark:


```

> Frame 58: 170 bytes on wire (1360 bits), 170 bytes captured (1360 bits) on interface \Device\NPF_{B1D78D4A-2FFB-4E5D-89A8-
> Ethernet II, Src: VMware_3d:23:3d (00:50:56:3d:23:3d), Dst: VMware_82:52:62 (00:0c:29:82:52:62)
> Internet Protocol Version 4, Src: 192.168.177.101, Dst: 192.168.177.100
> Transmission Control Protocol, Src Port: 49759, Dst Port: 9999, Seq: 1, Ack: 1, Len: 116
✓ Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Bind, Fragment: Single, FragLen: 116, Call: 2
  Version: 5
  Version (minor): 0
  Packet type: Bind (11)
  Packet Flags: 0x03
  Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
  Frag Length: 116
  Auth Length: 0
  Call ID: 2
  Max Xmit Frag: 5840
  Max Recv Frag: 5840
  Assoc Group: 0x00000000
  Num Ctx Items: 2
  Ctx Item[1]: Context ID:0, 12345678-1234-1234-1234-123456789abc, 32bit NDR
  Ctx Item[2]: Context ID:1, 12345678-1234-1234-1234-123456789abc, Bind Time Feature Negotiation

```

As mentioned earlier, we'll explore RPC over the network more deeply in a future section. But if you want a head start or need some background, feel free to check out my research [here](#).

2. Authentication Level: Connect

Now let's move on to the next level: `RPC_C_AUTHN_LEVEL_CONNECT`.

This level **authenticates only at the time of connection** — during the initial bind phase. Once the connection is established, no further identity checks are performed. That means **only the bind packet is authenticated**, and **subsequent remote procedure calls won't include authentication data**.

You can find a sample code for this scenario [here](#), applying all authentication functions discussed earlier

Just make sure to run the client with valid credentials for an authenticated Windows account, and update the IP address of the server accordingly.

428	220.540260	192.168.177.101	192.168.177.100	DCERPC	218 Bind: call_id: 2, Fragment: Single, 2 context items: 12345678-1234-1234-1234-123456789abc V1.0 (32bit NDR), 12345678-1234-1234-1234-123456789abc V1.0
429	220.553548	192.168.177.100	192.168.177.101	DCERPC	340 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2 results: Acceptance, Negotiate ACK, NTLMSSP_CHALLENGE
430	220.563712	192.168.177.101	192.168.177.100	DCERPC	488 AUTH3: call_id: 2, Fragment: Single, NTLMSSP_AUTH, User: CLIENT\dom
431	220.563895	192.168.177.101	192.168.177.100	DCERPC	109 Request: call_id: 2, Fragment: Single, opnum: 0, Ctx: 0 12345678-1234-1234-1234-123456789abc V1
433	220.590397	192.168.177.100	192.168.177.101	DCERPC	82 Response: call_id: 2, Fragment: Single, Ctx: 0 12345678-1234-1234-1234-123456789abc V1

Here's a capture of the network traffic when using **Connect-level authentication**:

- In **the second packet**, you'll see the server sends an **NTLM challenge**.
- In third **packet**, the client responds with its **NTLM response**.
- After that, the client proceeds with the remote procedure call.

Let's see the bind packet now


```

√ Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Bind, Fragment: Single, FragLen: 164, Call: 2
  Version: 5
  Version (minor): 0
  Packet type: Bind (11)
  > Packet Flags: 0x03
  > Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
    Frag Length: 164
    Auth Length: 40
    Call ID: 2
    Max Xmit Frag: 5840
    Max Recv Frag: 5840
    Assoc Group: 0x00000000
    Num Ctx Items: 2
  > Ctx Item[1]: Context ID:0, 12345678-1234-1234-1234-123456789abc, 32bit NDR
  > Ctx Item[2]: Context ID:1, 12345678-1234-1234-1234-123456789abc, Bind Time Feature Negotiation
√ Auth Info: NTLMSSP, Connect, AuthContextId(0)
  Auth type: NTLMSSP (10)
  Auth level: Connect (2)
  Auth pad len: 0
  Auth Rsvd: 0
  Auth Context ID: 0
  > NTLM Secure Service Provider

```

You'll see that the **authentication level** is set to **CONNECT** and the **Auth Length** header not equal to zero anymore.

Let's check now the function call

```

Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Request, Fragment: Single, FragLen: 55, Call: 2, Ctx:
  Version: 5
  Version (minor): 0
  Packet type: Request (0)
  Packet Flags: 0x03
  Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
  Frag Length: 55
  Auth Length: 0
  Call ID: 2
  Alloc hint: 31
  Context ID: 0
  Opnum: 0
  [Response in frame: 22]
  Stub data: 13000000000000001300000048656c6c6f2c20525043205365727665722100

```

As expected, there's **no authentication data** included.

This is how connect-level authentication works: identity verification happens once, and that's it.

Before we move on to the next authentication level, there's something important to clarify: *Even if we use this server which registers an authentication service, our old client without any authentication can still connect. That's what I mentioned earlier — just registering an authentication service on the server doesn't guarantee that clients will actually authenticate themselves.*

To enforce this, the server needs to implement additional security checks — otherwise, unauthenticated clients can still bind and make calls.

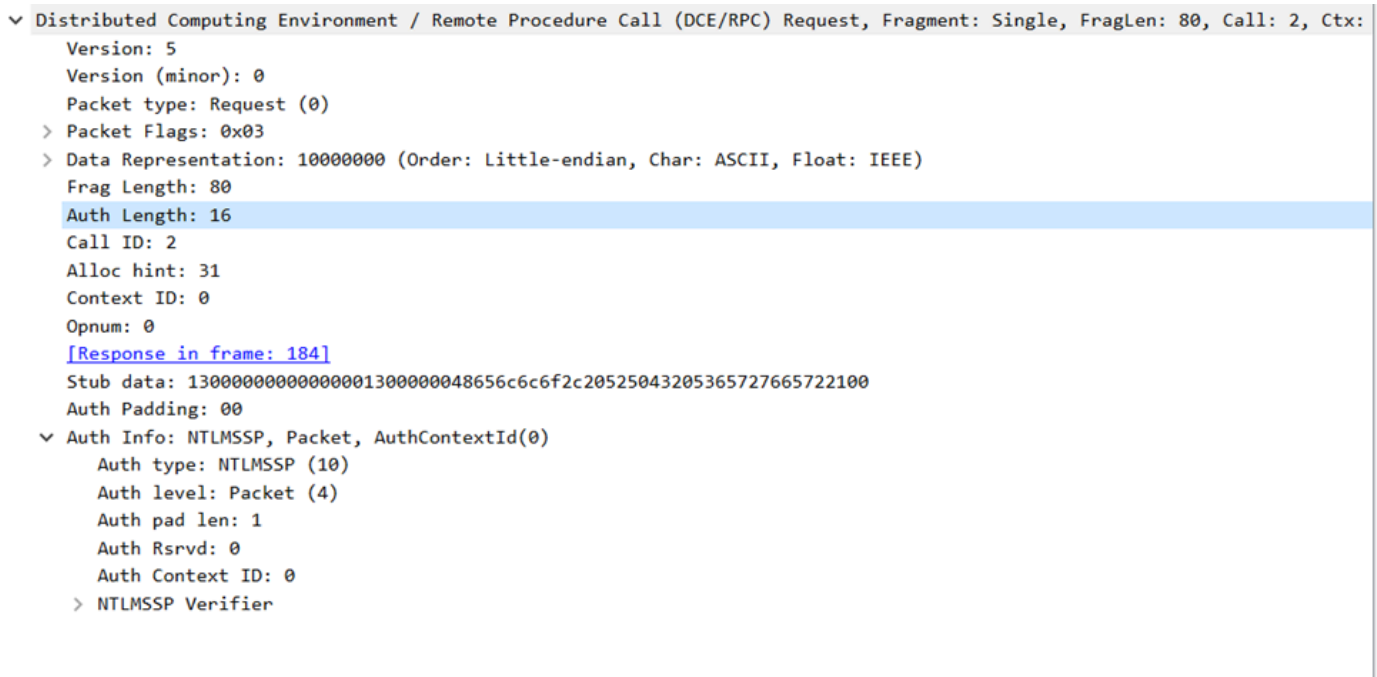
3. Authentication Level: Call

At this level, authentication happens **not only during the initial binding**, but also with **every remote procedure call**. This adds an extra layer of security compared to **CONNECT**, since the server can verify the client's identity on each call.

You can see this in action using the client and server example [here](#).

In wireshark, you'll notice that **function call now includes authentication data**, which confirms that authentication is being enforced at the **call level**.

Here's a screenshot showing what that looks like in the traffic:



```

Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Request, Fragment: Single, FragLen: 80, Call: 2, Ctx:
  Version: 5
  Version (minor): 0
  Packet type: Request (0)
  > Packet Flags: 0x03
  > Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
  Frag Length: 80
  Auth Length: 16
  Call ID: 2
  Alloc hint: 31
  Context ID: 0
  Opnum: 0
  [Response in frame: 184]
  Stub data: 13000000000000001300000048656c6c6f2c20525043205365727665722100
  Auth Padding: 00
  > Auth Info: NTLMSSP, Packet, AuthContextId(0)
    Auth type: NTLMSSP (10)
    Auth level: Packet (4)
    Auth pad len: 1
    Auth Rsvd: 0
    Auth Context ID: 0
    > NTLMSSP Verifier

```

4. Authentication Level: Packet Privacy

This is the highest level of authentication available in RPC. Like **CALL** level, authentication happens on **every remote procedure call**, but with a major upgrade:

At this level, **the entire RPC traffic is encrypted**, and **the integrity of each packet is also verified**.

You can check out a working example using this level of authentication [here](#).

After running the client and server, inspect the traffic in **Wireshark**.

```

▼ Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Request, Fragment: Single, FragLen: 144, Call: 2, Ctx: 0,
  Version: 5
  Version (minor): 0
  Packet type: Request (0)
  > Packet Flags: 0x03
  > Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
  Frag Length: 144
  Auth Length: 16
  Call ID: 2
  Alloc hint: 84
  Context ID: 0
  Opnum: 0
  [Response in frame: 1079]
  Encrypted stub data: 4b4816d8b747541abe4c40b5f8434a4ccaf2857e77b87e216b888a40bbf36ec29152335c5267ddad005079feaf42e139eebe0d
▼ Auth Info: NTLMSSP, Packet privacy, AuthContextId(0)
  Auth type: NTLMSSP (10)
  Auth level: Packet privacy (6)
  Auth pad len: 12
  Auth Rsvrd: 0
  Auth Context ID: 0
  > NTLMSSP Verifier

```

If you look at the packet containing the function call, you'll notice something new — an **encrypted section** labeled as:

Encrypted stub data

This indicates that the call's data is fully encrypted, unlike previous levels where the call arguments were visible in plaintext.

In our setup, **NTLM** is the protocol responsible for providing both encryption and integrity protection.

This process is referred to as:

- **Sealing** (for encryption), and
- **Signing** (for integrity checking)

If you look inside the **NTLM header** in the bind packet, you'll see that the client has requested both sealing and signing — which was not the case with the lower authentication levels we explored earlier.

Here's a screenshot from Wireshark that highlights this behavior:

```

NTLM Message Type: NTLMSSP_NEGOTIATE (0x00000001)
▼ Negotiate Flags: 0xe20882b7, Negotiate 56, Negotiate Key Exchange, Negotiate 128, Negotiate Version, Negotiate Extended
1... .. = Negotiate 56: Set
.1.. .. = Negotiate Key Exchange: Set
..1. .. = Negotiate 128: Set
...0 ... = Negotiate 0x10000000: Not set
.... 0... = Negotiate 0x08000000: Not set
.... .0.. = Negotiate 0x04000000: Not set
.... ..1. = Negotiate Version: Set
.... ...0 = Negotiate 0x01000000: Not set
.... ....0 = Negotiate Target Info: Not set
.... ....0.. = Request Non-NT Session Key: Not set
.... ......0 = Negotiate 0x00200000: Not set
.... .....0 = Negotiate Identify: Not set
.... .....1... = Negotiate Extended Session Security: Set
.... .....0.. = Negotiate 0x00040000: Not set
.... .....0. = Target Type Server: Not set
.... .....0 = Target Type Domain: Not set
.... .....1... = Negotiate Always Sign: Set
.... .....0.. = Negotiate 0x00004000: Not set
.... .....0. = Negotiate OEM Workstation Supplied: Not set
.... .....0 = Negotiate OEM Domain Supplied: Not set
.... .....0.. = Negotiate Anonymous: Not set
.... .....0.. = Negotiate 0x00000400: Not set
.... .....1. = Negotiate NTLM key: Set
.... .....0 = Negotiate 0x00000100: Not set
.... .....1... = Negotiate Lan Manager Key: Set
.... .....0.. = Negotiate Datagram: Not set
.... .....1. = Negotiate Seal: Set
.... .....1 = Negotiate Sign: Set
.... .....0... = Request 0x00000008: Not set
.... .....1.. = Request Target: Set
.... .....1. = Negotiate OEM: Set
.... .....1 = Negotiate UNICODE: Set

```

If you want to decrypt RPC traffic in Wireshark — and you're using NTLM as the authentication service (like in our case) — it's pretty straightforward. You just need to provide the client's password in Wireshark:

Edit → Preferences → Protocols → NTLMSSP → NT Password

Believe it or not, a lot of RPC servers today still expose interfaces that allow access with **no authentication** (aka, authentication level **none**). That's why it's always worth checking whether you can bind to an interface without creds — and then see what functions you're allowed to call.

One well-known example is the **Windows Netlogon interface**, which accepts unauthenticated binding. You can even call a few interesting functions once you're in. My tool [NauthNRPC](#), for instance, binds to the Netlogon interface with **RPC_C_AUTHN_LEVEL_NONE** and call some function to check whether a specific user exists in the domain. I prefer to stop today at this point. This is just the beginning of our dive into RPC security. There's a lot more to uncover, and we'll dig into even more interesting details in the upcoming parts. Hopefully, binding authentication is clear, see you in the next one!