Windows Inter Process Communication A Deep Dive Beyond the Surface - Part 5

sud0ru.ghost.io/windows-inter-process-communication-a-deep-dive-beyond-the-surface-part-5

Sud0Ru July 2, 2025

Jul 2, 2025 7 min read

Welcome to part 5 of the IPC series and part 4 of our deep dive into RPC.
In the previous post, we kicked off the topic of RPC security by exploring binding authentication and digging into the different authentication levels.

In today’s post, we’ll continue our journey by looking at more ways to secure RPC servers where we'll talk about how to secure your RPC interface

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.

So let's jump in

As I mentioned in Part 4, there are multiple ways to secure your RPC server. We've already looked in depth at binding authentication, the client-side mechanism for attaching identity to a binding handle, which is handled entirely by the RPC runtime. Now, let's continue exploring our security options. We'll divide them into two main categories: **securing the interface itself**, and **securing the endpoint** used to access that interface.

1. Securing the Interface

A. Registration Flags

When you register an RPC interface using the RpcServerRegisterIf2 function (or RpcServerRegisterIf3), you can apply certain flags to influence the behavior and access control of that interface. We've already worked with version 2 of this API — RpcServerRegisterIf2 — which supports several useful flags that play a big role in the security of your interface:

```
RPC_STATUS RpcServerRegisterIf2(
  RPC_IF_HANDLE      IfSpec,
  UUID               *MgrTypeUuid,
  RPC_MGR_EPV        *MgrEpv,
  unsigned int       Flags,
  unsigned int       MaxCalls,
  unsigned int       MaxRpcSize,
  RPC_IF_CALLBACK_FN *IfCallbackFn
);
```

Let's go over a few of the most interesting and relevant Flags values here:

RPC_IF_ALLOW_LOCAL_ONLY
This flag tells the RPC runtime to reject all remote client calls. Any request that uses a protocol sequence like ncacn_* or ncadg_*will be rejected. The only exception is ncacn_np (Named Pipes), which can still be used but only when accessed locally. Remember that named pipes can be reached either over the network (e.g., \\SRV\pipe\MyPipe) or locally (\\.\pipe\MyPipe), so this flag ensures that only the local version is allowed.

RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH
When this flag is set, the RPC runtime will *always* invoke the registered security callback, even if the client is unauthenticated. This means your callback function becomes your chance to inspect each call regardless of the client's identity or protocol and decide whether to allow or block it. This is useful if you want to implement custom logic like IP filtering or check additional runtime conditions like specific authentication level.

(We'll cover how to register these callbacks a bit later.)

RPC_IF_ALLOW_SECURE_ONLY
This flag enforces that clients *must* use an authentication level higher than RPC_C_AUTHN_LEVEL_NONE. So if the client tries to bind or call functions with no authentication at all, the RPC runtime will reject the request.

As u remember we saw that even when we registered authentication, the client was still able to connect without providing authentication (i.e., using authentication level **none**). This flag is one way to prevent that kind of behavior.

Something interesting about this flag too, Microsoft explicitly notes that:

> Specifying this flag allows clients to come through on the **NULL** session. On Windows XP and Windows Server 2003, such clients are not allowed.

What does that mean?

Let's say your RPC server uses Named Pipes (over SMB) as the transport. SMB itself has its own authentication mechanism — totally separate from RPC's. This means a client could connect over a "null session" (when u access the named pipes with empty creds), which bypasses authentication at the transport layer. In older versions of Windows, clients that connected over a NULL session were treated as authenticated. Consequently, even with the `RPC_IF_ALLOW_SECURE_ONLY` flag set and the authentication level at `NONE`, such clients were granted access. Newer versions fixed this issue: if you connect through a NULL session with authentication level `NONE`, the call now fails with an `RPC_S_ACCESS_DENIED` error.

Another key point: this flag only checks *whether* the client used authentication or not but does not imply or guarantee a high level of privilege on the part of the calling use. For example, even if the client binds with a guest account, it will pass the check as long as the authentication level isn't NONE.

RPC_IF_SEC_NO_CACHE
which is used to disables security callback caching. We will talk about caching later

B. Security Descriptor

`RpcServerRegisterIf3` lets you attach a discretionary access-control list (DACL) directly to an RPC interface.
Whenever a client invokes a procedure on that interface, the RPC runtime first checks the

caller's access token against this **security descriptor (SD)**; the call is dispatched only if the token satisfies the DACL.

```
RPC_STATUS RpcServerRegisterIf3(
    RPC_IF_HANDLE        IfSpec,
    UUID                *MgrTypeUuid,        // optional
    RPC_MGR_EPV         *MgrEpv,             // optional
    unsigned int         Flags,
    unsigned int         MaxCalls,
    unsigned int         MaxRpcSize,
    RPC_IF_CALLBACK_FN  *IfCallback,         // optional
    void                *SecurityDescriptor  // optional – pointer to SD
```

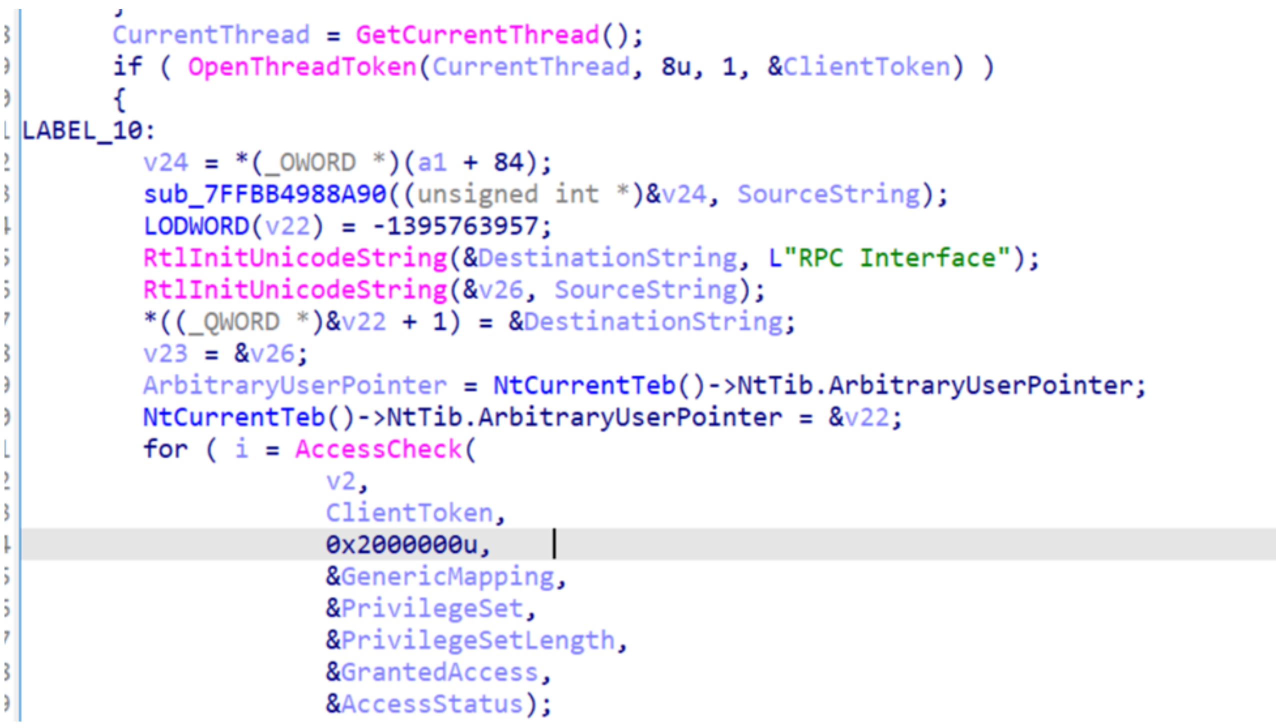
If you pass NULL for SecurityDescriptor, the RPC runtime applies its built-in default SD, which grants **CALL** access to:

- ANONYMOUS LOGON (S-1-5-7)
- Everyone (S-1-1-0)
- RESTRICTED (S-1-5-12)
- BUILTIN\Administrators
- SELF (the server process)

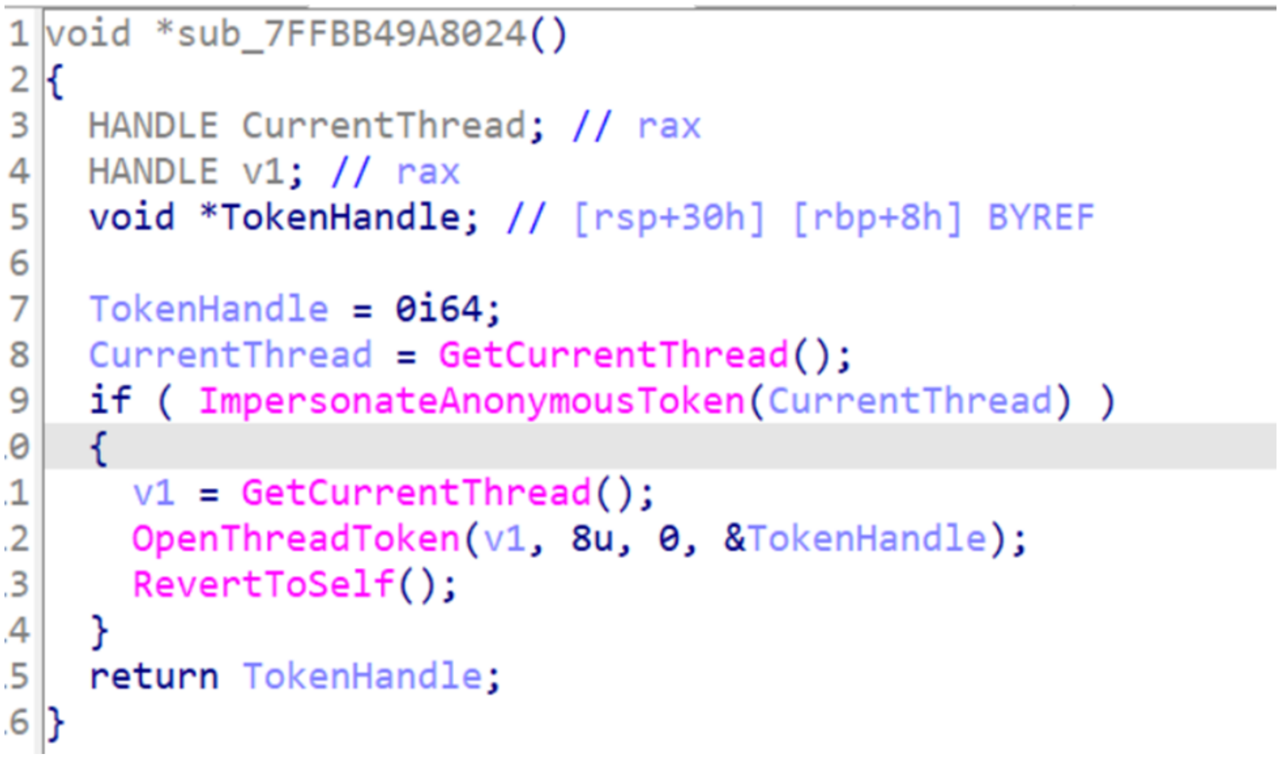
Now, where does the caller's token come from?

Transport / Auth	**How the token is obtained**	**Token seen by the SD**
ALPC (`ncalrpc`)	The kernel knows which process/thread owns the ALPC port.	Full local-user token.
Named pipe / SMB (`ncacn_np`)	SMB session setup (Kerberos, NTLM, guest, or null session) runs before the first RPC byte.	Token produced by SMB – usually the real network-user token.
TCP/UDP (`ncacn_ip_tcp`, `ncadg_ip_udp`) without `RpcBindingSetAuthInfo`	The socket layer has no concept of Windows logon. RPC therefore impersonates ANONYMOUS LOGON (user SID S-1-5-7)	Anonymous token.
Same TCP/UDP with `RpcBindingSetAuthInfo[Ex]`	The client adds RPC-level auth (Kerberos, NTLM, Schannel, etc.).	Token built from the validated credentials.

As you can see, ALPC and SMB are authenticated transports, whereas TCP is not. An interesting scenario arises when a client uses TCP with no RPC-level authentication: because TCP provides no built-in identity, the RPC runtime impersonates **ANONYMOUS LOGON** and uses that token when it checks the interface's security descriptor. This behavior can be confirmed by reversing the relevant code in **rpcrt4.dll**.



```
    CurrentThread = GetCurrentThread();
    if ( OpenThreadToken(CurrentThread, 8u, 1, &ClientToken) )
    {
LABEL_10:
      v24 = *(_OWORD *)(a1 + 84);
      sub_7FFBB4988A90((unsigned int *)&v24, SourceString);
      LODWORD(v22) = -1395763957;
      RtlInitUnicodeString(&DestinationString, L"RPC Interface");
      RtlInitUnicodeString(&v26, SourceString);
      *((_QWORD *)&v22 + 1) = &DestinationString;
      v23 = &v26;
      ArbitraryUserPointer = NtCurrentTeb()->NtTib.ArbitraryUserPointer;
      NtCurrentTeb()->NtTib.ArbitraryUserPointer = &v22;
      for ( i = AccessCheck(
                  v2,
                  ClientToken,
                  0x2000000u,
                  &GenericMapping,
                  &PrivilegeSet,
                  &PrivilegeSetLength,
                  &GrantedAccess,
                  &AccessStatus);
```



```
void *sub_7FFBB49A8024()
{
  HANDLE CurrentThread; // rax
  HANDLE v1; // rax
  void *TokenHandle; // [rsp+30h] [rbp+8h] BYREF

  TokenHandle = 0i64;
  CurrentThread = GetCurrentThread();
  if ( ImpersonateAnonymousToken(CurrentThread) )
  {
    v1 = GetCurrentThread();
    OpenThreadToken(v1, 8u, 0, &TokenHandle);
    RevertToSelf();
  }
  return TokenHandle;
}
```

As u can see from IDA screenshots inside RPC runtime in case of TCP access without authentication:

- The function tries to impersonate the "anonymous" user for the current thread.
- If impersonation is successful, it opens the thread's impersonation token.
- It then reverts the thread to its original state and returns a handle to the impersonation token.
- If impersonation fails, it simply returns 0 (indicating no token was retrieved).

After that the access check will be against the returned token.

Final not here, because the thread is reverted to its original state in the end, any attempt to impersonate the client in your RPC server code with `RpcImpersonateClient(...)` will fail with `RPC_S_BINDING_HAS_NO_AUTH` (1764)

C. Security callback:

As we see before you can register a security callback using `RpcServerRegisterIf2` or `RpcServerRegisterIf3`. This callback will be called when the client will be connected to RPC server
The security call back will look like this:

```
RPC_STATUS CALLBACK SecurityCallback(RPC_IF_HANDLE Interface, void*
pBindingHandle){
    printf("The security callback is called");
    return RPC_S_OK; // Whoever binds to the interface, we will allow the
connection
}
```

This callback function will be invoked when a call is made to the interface, although it will be called after the SD is checked. If the callback function returns *RPC_S_OK* then the call will be allowed, anything else will deny the call. The callback gets a pointer to the interface and the binding handle and can do various checks to determine if the caller is allowed to access the interface.

Using the security call back alone will make the RPC runtime rejected the unauthenticated clients by default unless you add the flag **`RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH`** which will allow the unauthenticated clients from accessing the interface and give the security call back the role of check their identity.

D. Security Callback Caching

When you register a **security callback**, it gets called during the security checks on client requests. If the callback succeeds (returns `RPC_S_OK`), the result may be cached so on subsequent calls from the same client, the RPC runtime can reuse the cached result instead of invoking the callback again.

This caching mechanism depends on a few important factors:

- **Caching is tied to the client's security context**, derived from the binding handle. If the server hasn't registered any authentication (or if the client hasn't set up authentication), then caching is disabled for that call.
- If you register the interface with the `RPC_IF_SEC_NO_CACHE` flag, the RPC runtime will always call the security callback on every client request, effectively **disabling caching**.
- There's also an undocumented flag, `RPC_IF_SEC_CACHE_PER_PROC`, which changes the caching behavior: instead of caching at the interface level, it caches per call. This means if the cache has a successful result for function X on interface TEST, a call to function Y on the same interface will still trigger the security callback again.

Caching is a very important aspect of interface security that you need to pay close attention to. One vulnerability discovered by Akamai involved a security callback registered on a Windows service's RPC interface. This security callback was meant to restrict certain functions for remote clients, while still allowing other functions to be called remotely.

However, if an attacker first called one of the **allowed functions**, the positive result of the security callback would be **cached**. Under certain circumstances, as demonstrated by Akamai, this cached result could then be abused to call **restricted functions**, bypassing the intended security controls.

We'll dive deeper into caching and Akamai's research when we cover **attacking RPC** later in this series.

I'll wrap up this part here, as we've now covered interface security. I hope everything is clear so far. We still have more security measures to discuss in upcoming parts.

But before we get to those, in the next post I'll show you how to use **Impacket** for RPC security research, including things you can't do with native programming, and how to set up your environment for the more advanced, tricky parts.

Until then, see you soon!