kaspersky

A journey into forgotten Null Session and MS-RPC interfaces

Haidar Kabibo

Contents

About null session 4
Enumerating network interfaces without authentication 5
Null session vs authentication level 7
No-Auth working despite the policy? 9
Methodology 10
A. Enumerating RPC endpoints 10
1. Using endpoint mapper 10
2. Using Nmap ports scan 11
B. Collecting interface UUIDs 12
C. Brute-forcing OPNUMs: 13
D. Case study: IObjectExporter interface 14
No-Auth against the MS-NRPC protocol 17
MS-NRPC functions 20
Enumerate Domain users and computers 20
Enumerate trusted domains 24
Conclusion 27

A journey into the forgotten Null Session and MS-RPC interfaces

It has been almost 24 years since the null session vulnerability was discovered. In the old days, it was possible to access the SMB named pipes using empty credentials and start to collect domain information. The best-known use of null sessions was to gather domain users through techniques such as RID (Relative Identifier) enumeration. RIDs uniquely identify users, groups, computers, and other entities within a domain. To enumerate them, the attacker manipulated MS-RPC interfaces to make some calls and collect information from the remote host. This process often follows brute-forcing, where simple passwords were tried against these users in an attempt to gain access to the system.

To prevent such attacks, Microsoft restricted null session capabilities by limiting what an attacker can do after connecting to the named pipes. However, to this day, the null session still exists and is enabled by default on Windows servers acting as domain controllers. The reason for its persistence may be legacy compatibility with other systems. On the other hand, Microsoft provides security policies that can be used to stop all null session activity.

Even though null sessions still exists with limited capabilities, it's hard to find it enabled on domain controllers these days. Most system administrators close null sessions by hardening the security policies and monitoring activity on domain controllers. This includes monitoring anonymous access through SMB. As penetration testers, we always ask: “Is it as secure as it seems?” In this case, we asked if we can bypass policies and restrictions today, after 24 years, and bring the idea of anonymous access back to life.

This article is tailored for security researchers and penetration testers seeking to enhance their understanding of MS-RPC interfaces and refine their research techniques. At the end of the article, there is a link to a tool that implements all the enumeration techniques discussed. It's important to note that all information in this article is intended for legitimate security research purposes only and must not be used for illegal activities.

This article is in two parts. In the first part, I will take you on a journey through my security research methodology against MS-RPC interfaces, which I developed after observing some interesting behavior from one of the Windows interfaces. As a result of this research, I will show how we can link this behavior to null sessions. Putting it all together, I will demonstrate how we can revive the legacy of null

sessions by enumerating information from the domain controller and, most importantly, enumerating domain users without triggering any alerts and without being stopped by defenders.

In the second part, I will delve deeper into the security layers of MS-RPC interfaces. In addition, I will analyze why this type of enumeration cannot be effectively halted, providing examples related to some Windows services. I will also show how the native Windows audit policy fails to effectively monitor such enumerations, and explore alternative methods that can be used.

About null session

Null sessions have emerged as a pivotal area of interest and concern within the field of cybersecurity.

A null session appears when access to a network resource, most commonly the IPC$ "Windows Named Pipe" share, is granted without authentication, more specifically, with empty credentials. IPC$ (Inter-Process Communication) is, as its name suggests, a hidden share that allows processes on different computers to communicate with each other. After gaining access, the attacker can bind one of the MS-RPC interfaces exposed by a particular named pipe inside IPC$ share and start gathering information such as shares, users, groups, registry keys and much more.

In newer versions of Windows, the null session capability has become more restricted, and is only available on Windows servers that act as domain controllers. When you upgrade your server to a domain controller, only the following named pipes can be accessed through a null session by default: "`\pipe\netlogon`", "`\pipe\samr`", and "`\pipe\lsarpc`".

To prevent null sessions, two related system policies were introduced: Restrict anonymous access to Named Pipes and Shares and Network access: Named Pipes that can be accessed anonymously. The first policy, Restrict anonymous access to Named Pipes and Shares, is enabled by default. The second policy, Network access: Named Pipes that can be accessed anonymously, contains the three named pipes discussed above. To prevent any action related to null sessions, domain administrators usually set the latter policy to empty, so that these named pipes cannot be accessed anonymously.

Enumerating network interfaces without authentication

During my work in traffic analysis, I noticed many packets related to DCOM communications between domain controller and other endpoints, which were tagged by Wireshark under `IOXIDResolver` RPC interface and `ServerAlive2()` method. The `IOXIDResolver` interface is actually the `IObjectExporter` interface. As Microsoft mentioned, it is used for OXID resolution, pinging and server aliveness testing. In brief, it helps in the process of locating and connecting to remote objects involved in resolving OXID references to actual network locations (such as IP addresses) of objects in a DCOM environment.

One of the `IObjectExporter` methods is `ServerAlive2`. The `ServerAlive2 (OPNUM 5)` method was introduced with version 5.6 of the DCOM Remote Protocol, and extends the `ServerAlive` method. It returns string and security bindings for the object resolver, allowing the client to choose the most appropriate settings that are compatible with both client and server. The `IObjectExporter` interface uses TCP port 135 as its MS-RPC endpoint.

Returning to traffic analysis, when I examined the response packet for `ServerAlive2()`, I saw all the network interfaces for the domain controller in it. The surprise was the absence of RPC authentication.

In the traffic for each TCP stream related to DCOM communication, there were four packets. The first one is related to binding the `IObjectExporter` interface, the second is related to the server binding response, the third is related to the `ServerAlive2` function call, and the last packet is the response for the `ServerAlive2` function.

Note that if you are using native Windows APIs, the process of creating any DCOM object should start with the `IObjectExporter` interface.

```
v Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Bind, Fragment: Single,
    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
```

Bind request packet

The image above shows a bind request for the `IObjectExporter` interface. You can see that the Auth Length header is zero, indicating that the authentication level is None, so there is no authentication. With just two packets from the client, we can enumerate network interfaces for the remote host without authentication.

I searched about this kind of no authentication against the `IObjectExporter` interface and saw this article by Airbus about remote enumeration of network interfaces without any authentication.

The idea struck me: what if there are other RPC interfaces vulnerable to no authentication, what kind of enumeration could be obtained from them? Can we map it to the famous null session? And what research strategy should I follow to find out?

Null session vs authentication level

Before delving into the research strategy, I want to highlight some key concepts related to MS-RPC security. There are seven authentication levels in MS-RPC.

```
#define RPC_C_AUTHN_LEVEL_DEFAULT       0
#define RPC_C_AUTHN_LEVEL_NONE          1
#define RPC_C_AUTHN_LEVEL_CONNECT       2
#define RPC_C_AUTHN_LEVEL_CALL          3
#define RPC_C_AUTHN_LEVEL_PKT           4
#define RPC_C_AUTHN_LEVEL_PKT_INTEGRITY 5
#define RPC_C_AUTHN_LEVEL_PKT_PRIVACY   6
```

RPC authentication levels

The first is the default authentication level for the specified authentication service. AUTHN_LEVEL_NONE implies no authentication, while AUTHN_LEVEL_PKT_PRIVACY indicates a high level of authentication that includes encryption, authentication, and integrity.

It's important to understand that having an authentication level set to NONE and having a null session are not the same thing. When we refer to a null session, we mean accessing IPC$ without authorization. In this scenario, we're talking about accessing a specific transport layer used for MS-RPC communication. Even if you manage to access IPC$ and open some pipes, you still need to authenticate the interface binding request. However, if you're using a TCP port as the MS-RPC endpoint, you don't have to worry about null sessions or IPC$. All you need to focus on is the interface authentication level (binding authentication).

We mentioned earlier the policies used to prevent null sessions. But is there a policy specifically designed to prevent no authentication against MS-RPC interfaces? The answer is yes. There is a security policy called the Restrict Unauthenticated RPC Clients policy. This policy is not configured by default. It can take three values:

- "None" allows all RPC clients to connect to RPC servers running on the machine where the policy setting is applied.
- "Authenticated" allows only authenticated RPC clients to connect to RPC servers running on the machine where the policy setting is applied. Exceptions are granted to interfaces that have requested them.
- "Authenticated without exceptions" allows only authenticated RPC clients to connect to RPC servers running on the machine where the policy setting is applied. No exceptions are allowed.

Usually, to prevent access to interfaces without authentication, this policy is set to "Authenticated." We will explain in Part 2 of this research why "Authenticated" is chosen over "Authenticated without exceptions."

No-Auth working despite the policy?

So far, we've discussed two ways to access MS-RPC interfaces:

- **IPC$ share**: This involves using named pipes. I'll call it the Null Session path. However, this path is typically closed off by system administrators through the policies we mentioned earlier.
- **TCP port as endpoint**: This path utilizes TCP ports such as 135 or high TCP dynamic ports. Let's call it the No-Auth path. Similarly, this path is often restricted by system administrators through the "Restrict Unauthenticated RPC Clients" policy.

The goal was to find a way to access the MS-RPC interfaces even if the Null Session path was closed. However, attempting to use the No-Auth path would run into the "Restrict Unauthenticated RPC Clients" policy, which would thwart the attempt. However, even with this policy in place, enumeration of network interfaces using the `IObjectExporter` interface seems to function properly. This raises the question: why is it working despite the policy? Additionally, are there other interfaces that exhibit similar behavior?

It's now time to delve into these questions and uncover the answers we're looking for.

Methodology

In this phase, we will discuss each step we took in order to achieve our goal. Don't forget that we are looking for MS-RPC interfaces that we can access without authentication (authentication level equal to one) in restricted systems. So first, let me give you with more information about our target system.

We will be using Windows Server 2022 as a domain controller, and the following policies will be applied:

- No null session through IPC$:
  - "Restrict anonymous access to Named Pipes and Shares" is enabled
  - "Network access: Named Pipes that can be accessed anonymously" is not defined.
- "Restrict Unauthenticated RPC Clients" policy is set to Authenticated.

These policies are applied by all system administrators to prevent any activity related to a null session and accessing MS-RPC interfaces without authentication.

A. Enumerating RPC endpoints

As we saw earlier, in the RPC environment, interfaces can be accessed through various types of endpoints. The initial research step was to gather as many endpoints as possible using different approaches.

1. **Using endpoint mapper**

The first tool I used for endpoint enumeration was Impacket, which is a collection of Python classes for working with network protocols. Impacket focuses on providing low-level programmatic access to the packets and, for some protocols (e.g., SMB1-3 and MSRPC), the protocol implementation itself.

Impacket provides two scripts for enumerating RPC interfaces and endpoints: `rpcdump.py` and `rpcmap.py`. In this section, I will focus on the first one.

`rpcdump.py` is used to dump information about remote RPC endpoints via the `epmapper` (endpoint mapper) interface. The internal work of this script is simple: it just binds the `epmapper` interface. Endpoint mapper is a service on a remote procedure call (RPC) server that maintains a database of dynamic endpoints and allows clients to map an interface/object UUID pair to a local dynamic endpoint. In short, after binding this interface and calling a lookup method (OPNUM 2), all the endpoints in the remote host will be returned.

The screenshot below shows a portion of the `rpcdump.py` output.

```
156 Protocol: [MS-DRSR]: Directory Replication Service (DRS) Remote Protocol
157 Provider: ntdsai.dll
158 UUID     : E3514235-4B06-11D1-AB04-00C04FC2DCD2 v4.0 MS NT Directory DRS Interface
159 Bindings:
160           ncacn_np:\\WIN-SO9H290I5NL[\pipe\b9459c55c28cac8a]
161           ncacn_http:192.168.177.177[49670]
162           ncalrpc:[NTDS_LPC]
163           ncalrpc:[OLE395259F93F9D9566C47334E3284B]
164           ncacn_ip_tcp:192.168.177.177[49668]
165           ncacn_ip_tcp:192.168.177.177[49664]
166           ncalrpc:[samss lpc]
167           ncalrpc:[SidKey Local End Point]
168           ncalrpc:[protected_storage]
169           ncalrpc:[lsasspirpc]
170           ncalrpc:[lsapolicylookup]
171           ncalrpc:[LSA_EAS_ENDPOINT]
172           ncalrpc:[lsacap]
173           ncalrpc:[LSARPC_ENDPOINT]
174           ncalrpc:[securityevent]
175           ncalrpc:[audit]
176           ncacn_np:\\WIN-SO9H290I5NL[\pipe\lsass]
```

Endpoint mapper database

`ncacn_ip_tcp` refers to TCP. All endpoints tagged with this field can be added to the endpoint list. However, `rpcdump` won't provide a comprehensive list of all endpoints on the remote host because some endpoints, known as "Well-known Endpoints," do not need to register their interfaces with the endpoint mapper. For this reason, other approaches should be used.

2. **Using Nmap ports scan**

Actually, this stage can serve as an alternative to the previous one. We are now focusing on endpoints that are related to TCP ports. By conducting a full port scan using tools like Nmap, we can obtain this information. However, I also wanted to demonstrate the use of the endpoint mapper service because it can be helpful in other scenarios.

The Nmap scan returns all available ports, their states and the default services running on those ports. The image below shows many high ports that can be used in our endpoints list.

```
└─# nmap -p- -n 192.168.177.177 --min-rate=10000
Starting Nmap 7.94 ( https://nmap.org ) at 2024-03-25 09:19 EDT
Nmap scan report for 192.168.177.177
Host is up (0.00100s latency).
Not shown: 65515 filtered tcp ports (no-response)
PORT      STATE SERVICE
53/tcp    open  domain
88/tcp    open  kerberos-sec
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
389/tcp   open  ldap
445/tcp   open  microsoft-ds
464/tcp   open  kpasswd5
593/tcp   open  http-rpc-epmap
636/tcp   open  ldapssl
3268/tcp  open  globalcatLDAP
3269/tcp  open  globalcatLDAPssl
5985/tcp  open  wsman
9389/tcp  open  adws
49664/tcp open  unknown
49667/tcp open  unknown
49668/tcp open  unknown
49670/tcp open  unknown
49671/tcp open  unknown
49681/tcp open  unknown
49689/tcp open  unknown
```

Nmap full port scan

If you have access to the host, you can also use the netstat tool to enumerate TCP ports.

B. Collecting interface UUIDs

Impacket has another script called `rpcmap.py` that is used to enumerate all interfaces inside an endpoint. It uses the `MGMT` interface, which is responsible for getting interface UUIDs in a specific endpoint. Unfortunately, we will encounter a problem when using the `MGMT` interface. As you know, our research is intended to be done without providing any credentials, which means we're looking for interfaces that can be accessed without authentication. Unfortunately, the `MGMT` interface does not fall into this category. To address this issue, we have two possible solutions:

- Provide valid credentials to `rpcmap`.
- Modify `rpcmap's` internal workflow to brute-force UUIDs using its database without directly calling the `MGMT` interface.

For more reliable results I used valid credentials.

The image below shows example of using the `rpcmap.py` script to enumerate all interfaces related to the port 135 endpoint.

```
└─$ impacket-rpcmap ncacn_ip_tcp:192.168.177.177[135] -auth-rpc Administrator:Asd123456#
Impacket v0.11.0 - Copyright 2023 Fortra

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

Protocol: [MS-DCOM]: Distributed Component Object Model (DCOM) Remote
Provider: rpcss.dll
UUID: 4D9F4AB8-7D1C-11CF-861E-0020AF6E7C57 v0.0

Procotol: N/A
Provider: rpcss.dll
UUID: 64FE0B7F-9EF5-4553-A7DB-9A1975777554 v1.0

Protocol: [MS-DCOM]: Distributed Component Object Model (DCOM) Remote
Provider: rpcss.dll
UUID: 99FCFEC4-5260-101B-BBCB-00AA0021347A v0.0
```

Interfaces that can be accessed through port 135

At this point, I have identified all interfaces in the accessible TCP ports.

C. Brute-forcing OPNUMs:

Another use of `rpcmap.py` is to brute-force the OPNUMs for a specific interface. An OPNUM is an integer that corresponds to a particular method within an interface, and because we are looking for interfaces that can be accessed without authentication, we should use -auth-level 1 with `rpcmap` to use an authentication level equal to one (NONE).

The image below shows an example of using `rpcmap.py` to brute-force OPNUMs for a specific interface.

```
└─$ impacket-rpcmap ncacn_ip_tcp:192.168.177.177[135] -uuid '99fcfec4-5260-101b-bbcb-00aa0021347a v0.0' -brute-opnums -auth-level 1
Impacket v0.11.0 - Copyright 2023 Fortra

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)
```

Using `rpcmap.py` to brute-force OPNUMs for a specific interface

Here we encounter another issue related to the `MGMT` interface. Before brute-forcing any OPNUMs against an interface, `rpcmap` typically calls the `MGMT` interface in the endpoint to ensure the interface

can actually be accessed through that endpoint. However, this process poses a problem for us, especially since we can't resolve it by using credentials. During the brute-force stage, we need to use an authentication level equal to one.

To address this, we'll need to make some adjustments to the `rpcmap` workflow. Specifically, we'll need to prevent it from binding to the `MGMT` interface at the beginning of the brute-force process. This adjustment will allow us to proceed with our research effectively.

In the `rpcmap.py` OPNUM brute-force output, if an OPNUM falls under `rpc_x_bad_stub_data`, it indicates that incorrect arguments were passed to the function associated with that OPNUM. If OPNUM shows `nca_s_op_rng_error`, it means that it is not implemented. If OPNUM shows `success`, it indicates that the corresponding method was called successfully on the remote host.

Furthermore, if OPNUM falls under `rpc_access_denied`, it indicates that access to these methods is restricted without the appropriate credentials.

In the final step of the research, we can attempt to bind and brute-force OPNUMs for each UUID collected from the previous steps using `rpcmap.py`. For interfaces where we received `rpc_x_bad_stub_data` or `success`, we can say that the methods within these interfaces are vulnerable to no authentication.

D. Case study: IObjectExporter interface

For this stage, let's try to get the same results I got during traffic analysis (enumerating network interfaces without authentication using `IObjectExporter`), but using our methodology.

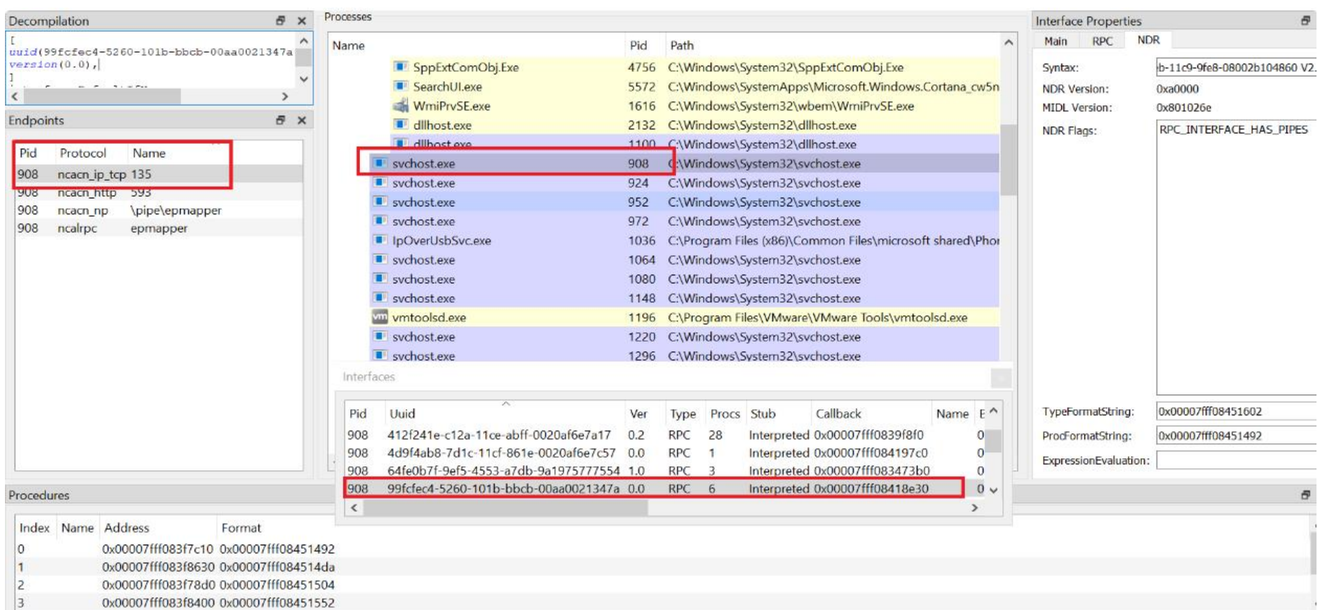
Let's start by brute-forcing OPNUMs against the UUID associated with the `IObjectExporter` interface. The interface is accessed through TCP port 135 with the UUID: `99fcfec4-5260-101b-bbcb-00aa0021347a`.

In the image provided in the previous section, you can see that OPNUMs 0, 1, 2, and 4 fall under `rpc_x_bad_stub_data`, indicating that incorrect arguments were passed to the functions associated with those OPNUMs. OPNUMs ranging from 6 to 64 show a range error, signifying that they are not implemented. Both OPNUM 3 and OPNUM 5 return `success`, indicating that the corresponding methods were called successfully on the remote host.

At this point if we sniff the traffic using Wireshark during the brute-forcing process and filter for OPNUM 5 (`ServerAlive2`), we can see the remote server's network interfaces in the response packet. However,

before we proceed, let me show you a useful tool that can be used to get information about the `IObjectExporter` interface, but in the target system itself.

The tool is useful for any research related to MS-RPC. It is called RPCView, is free, and allows us to explore and decompile all the RPC functionalities available on a Microsoft system. I will use it to compare the results obtained from the `rpcmap.py` script with the actual methods under the `IObjectExporter` interface on the remote host. RPCView can list all the methods within a specific interface. In addition, the tool allows you to generate a code that outlines the definition of these methods.



RPCView for *the* `IObjectExporter` interface

In the image above, you can see that the endpoint on TCP port 135 is running under `svchost.exe` and is associated with `rpcss.dll`. One of the interfaces under this endpoint is `IObjectExporter`. RPCView shows that `IObjectExporter` has six methods that were previously identified using `rpcmap.py`.

The Decompilation tab shows the declarations of each method (Proc). For the Proc5 (the method under the OPNUM 5), it looks like this:

```
[
uuid(99fcfec4-5260-101b-bbcb-00aa0021347a),
version(0.0),
```

```
]
interface DefaultIfName
{

error_status_t Proc5(
  [out]struct Struct_174_t* arg_2,
  [out][ref]struct Struct_68_t** arg_3,
  [out]long *arg_4);
}
```

The declaration indicates that the method can be invoked without any arguments, unlike most of the other methods. The only other method that doesn't necessarily need arguments is Proc3. The similarity between Proc3 and Proc5 in the declarations aligns with the findings from `rpcmap.py for` OPNUM 3 and OPNUM 5, respectively. This consistency confirms the accuracy of the enumeration results obtained through `rpcmap.py`.

Now, let's examine the network packets related to OPNUM 5 (ServerAlive2) using Wireshark. In the image below, Wireshark successfully identifies the request and classifies it as ServerAlive2.

oxid.opnum == 5

No.	Time	Source	Destination	Protocol	Length	Destination Port	Sec	Info
74	0.047211681	192.168.177.111	192.168.177.177	IOXIDResolver	90	135	73	ServerAlive2 request IOXIDResolver V0
75	0.048056022	192.168.177.177	192.168.177.111	IOXIDResolver	230	37896	61	ServerAlive2 response[Long frame (2 bytes)]

ServerAlive2() method through Wireshark

We can also observe the corresponding response from the server to the ServerAlive2() request.

```
Address: STRINGBINDINGs=2, SECURITYBINDINGs=7
  NumEntries: 57
  SecurityOffset: 35
  StringBinding[1]: TowerId=NCACN_IP_TCP, NetworkAddr="WIN-SO9H290I5NL"
  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=""
```

ServerAlive2() response *via* Wireshark

As you can see in the image, all network interfaces related to the remote host have been identified successfully.

No-Auth against the MS-NRPC protocol

Now that we've demonstrated how to map a no authentication against an RPC interface, let's proceed with our research and try to find interfaces that are vulnerable to No-Auth. At this point, we made a list of UUIDs for each endpoint.

Since `rpcmap.py` has no options to read UUIDs or endpoints from files, a small Bash script comes in handy.

After the script completes its execution, an interesting UUID is identified. This UUID is accessible through many high dynamic TCP ports, one of which is 49664.

The UUID belongs to the MS-NRPC protocol, which is according to Microsoft:

> *The Netlogon Remote Protocol is a remote procedure call (RPC) interface that is used for user and machine authentication on domain-based networks. The Netlogon Remote Protocol RPC interface is also used to replicate the database for backup domain controllers (BDCs).*
>
> *This protocol SHOULD use the following RPC protocol sequences, RPC over TCP/IP using high dynamic TCP port or RPC over named \PIPE\NETLOGON*

The Netlogon Remote Protocol was linked to the notorious Zerologon vulnerability.

This UUID will be displayed during brute-forcing attempts:

```
└─$ impacket-rpcmap ncacn_ip_tcp:192.168.177.177[49664] -uuid '12345678-1234-ABCD-EF00-01234567CFFB v1.0' -brute-opnums -auth-level 1
Impacket v0.11.0 - Copyright 2023 Fortra

Opnum 0: rpc_x_bad_stub_data
Opnum 1: rpc_x_bad_stub_data
Opnum 2: rpc_x_bad_stub_data
Opnum 3: rpc_x_bad_stub_data
Opnum 4: rpc_x_bad_stub_data
Opnum 5: rpc_x_bad_stub_data
Opnum 6: rpc_x_bad_stub_data
Opnum 7: rpc_x_bad_stub_data
Opnum 8: rpc_x_bad_stub_data
Opnum 9: rpc_x_bad_stub_data
Opnum 10: rpc_x_bad_stub_data
Opnum 11: rpc_x_bad_stub_data
Opnum 12: rpc_x_bad_stub_data
Opnum 13: rpc_x_bad_stub_data
Opnum 14: rpc_x_bad_stub_data
Opnum 15: rpc_x_bad_stub_data
Opnum 16: rpc_x_bad_stub_data
Opnum 17: rpc_x_bad_stub_data
Opnum 18: rpc_x_bad_stub_data
Opnum 19: rpc_x_bad_stub_data
Opnum 20: rpc_x_bad_stub_data
Opnum 21: rpc_x_bad_stub_data
Opnum 22: rpc_s_access_denied
Opnum 23: rpc_s_access_denied
Opnum 24: rpc_x_bad_stub_data
Opnum 25: rpc_x_bad_stub_data
Opnum 26: rpc_x_bad_stub_data
Opnum 27: rpc_x_bad_stub_data
Opnum 28: rpc_x_bad_stub_data
```

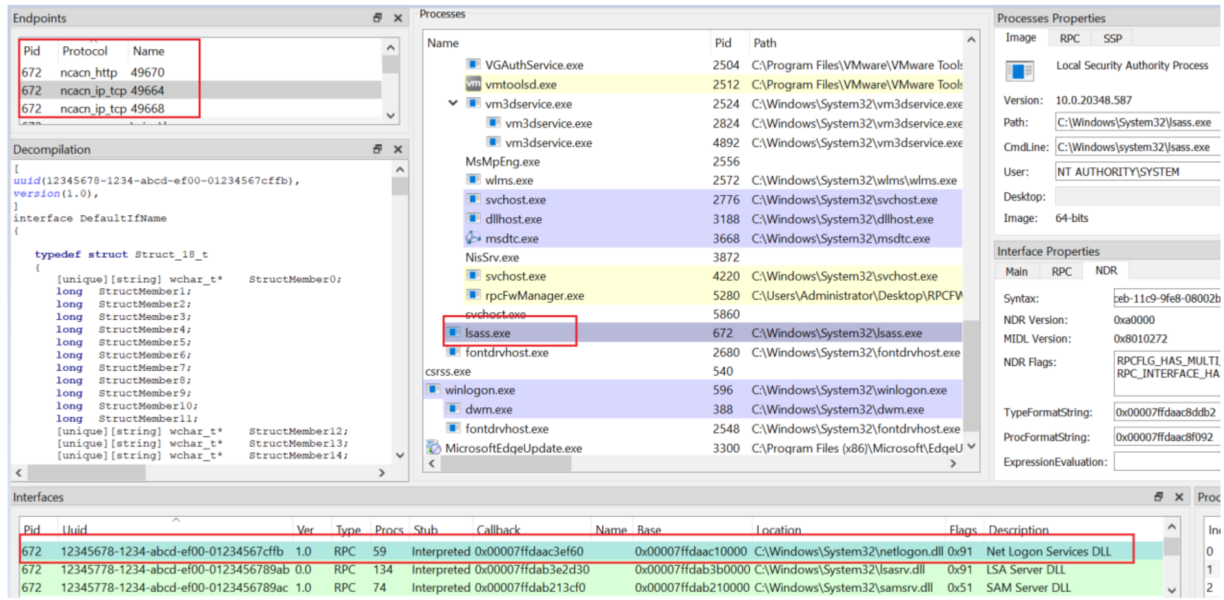
Brute-force against *the* MS-NRPC interface's OPNUMs

The `rpcmap` results:

- Almost all OPNUMs from 0 to 49 show `rpc_x_bad_stub_data`, meaning they were successfully called but with bad arguments.
- OPNUMs from 49 to 59 show access denied, and from 59 to 64 not implemented.

A portion of these results can be seen in the image above.

At this point, the research replicates the steps taken earlier with the `IObjectExporter` interface, starting with the use of the RPCview tool on the remote machine.



MS-NRPC interface through RPCview

As shown in the image above, the UUID is associated with `netlogon.dll`, operates under `lsass.exe` and has 59 methods.

Now, let’s shift focus to the network level and inspect the packets using Wireshark. Upon opening Wireshark, it's worth noting that it identifies the protocol under the name RPC_NETLOGON. It also recognizes all the methods under this interface.

```
11 0.001242492  192.168.177.111  192.168.177.177  DCERPC       138 49664  1 Bind: call_id: 1, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (32bit NDR)
12 0.001725563  192.168.177.177  192.168.177.111  DCERPC       126 54856  1 Bind_ack: call_id: 1, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance
14 0.003567293  192.168.177.111  192.168.177.177  RPC_NETLOGON  90 49664 73 NetrLogonUasLogon request[Malformed Packet] RPC_NETLOGON V1
15 0.004109104  192.168.177.177  192.168.177.111  DCERPC        98 54856 61 Fault: call_id: 1, Fragment: Single, Ctx: 0, status: nca_s_fault_ndr
23 0.005649215  192.168.177.111  192.168.177.177  DCERPC       138 49664  1 Bind: call_id: 2, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (32bit NDR)
24 0.005875897  192.168.177.177  192.168.177.111  DCERPC       126 54862  1 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance
26 0.007113244  192.168.177.111  192.168.177.177  RPC_NETLOGON  90 49664 73 NetrLogonUasLogoff request[Malformed Packet] RPC_NETLOGON V1
27 0.007883494  192.168.177.177  192.168.177.111  DCERPC        98 54862 61 Fault: call_id: 2, Fragment: Single, Ctx: 0, status: nca_s_fault_ndr
35 0.009843732  192.168.177.111  192.168.177.177  DCERPC       138 49664  1 Bind: call_id: 3, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (32bit NDR)
36 0.010193248  192.168.177.177  192.168.177.111  DCERPC       126 54866  1 Bind_ack: call_id: 3, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance
38 0.012252409  192.168.177.111  192.168.177.177  RPC_NETLOGON  90 49664 73 NetrLogonSamLogon request[Malformed Packet] RPC_NETLOGON V1
39 0.012954758  192.168.177.177  192.168.177.111  DCERPC        98 54866 61 Fault: call_id: 3, Fragment: Single, Ctx: 0, status: nca_s_fault_ndr
47 0.015973649  192.168.177.111  192.168.177.177  DCERPC       138 49664  1 Bind: call_id: 4, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (32bit NDR)
48 0.016520337  192.168.177.177  192.168.177.111  DCERPC       126 54882  1 Bind_ack: call_id: 4, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance
50 0.020609240  192.168.177.111  192.168.177.177  RPC_NETLOGON  90 49664 73 NetrLogonSamLogoff request[Malformed Packet] RPC_NETLOGON V1
51 0.022273117  192.168.177.177  192.168.177.111  DCERPC        98 54882 61 Fault: call_id: 4, Fragment: Single, Ctx: 0, status: nca_s_fault_ndr
59 0.023998195  192.168.177.111  192.168.177.177  DCERPC       138 49664  1 Bind: call_id: 5, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (32bit NDR)
60 0.024469233  192.168.177.177  192.168.177.111  DCERPC       126 54892  1 Bind_ack: call_id: 5, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance
62 0.026689505  192.168.177.111  192.168.177.177  RPC_NETLOGON  90 49664 73 NetrServerReqChallenge request[Malformed Packet] RPC_NETLOGON V1
63 0.027699311  192.168.177.177  192.168.177.111  DCERPC        98 54892 61 Fault: call_id: 5, Fragment: Single, Ctx: 0, status: nca_s_fault_ndr
71 0.029299561  192.168.177.111  192.168.177.177  DCERPC       138 49664  1 Bind: call_id: 6, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (32bit NDR)
72 0.029740200  192.168.177.177  192.168.177.111  DCERPC       126 54894  1 Bind_ack: call_id: 6, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance
74 0.030710592  192.168.177.111  192.168.177.177  RPC_NETLOGON  90 49664 73 NetrServerAuthenticate request[Malformed Packet] RPC_NETLOGON V1
75 0.031768731  192.168.177.177  192.168.177.111  DCERPC        98 54894 61 Fault: call_id: 6, Fragment: Single, Ctx: 0, status: nca_s_fault_ndr
83 0.035521267  192.168.177.111  192.168.177.177  DCERPC       138 49664  1 Bind: call_id: 7, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (32bit NDR)
```

MS-NRPC through Wireshark

MS-NRPC functions

Now it's time to find and use functions that do not require authentication. It's worth noting that there are functions that can be used to establish a connection and it's normal to access them without authentication, so they're not of interest to us. We'll use the `nrpc.py` library within Impacket, which handles the MS-NRPC protocol and contains most of the necessary functions. I started calling each function in this interface to see what information we can collect from the domain controller, and listed the most useful functions for gathering information about the domain controller into a table.

OPNUM	FUNCTION	OPNUM	FUNCTION
10	NetrGetDCName	34	DsrGetDcNameEx2
11	NetrGetAnyDCName	36	NetrEnumerateTrustedDomainsEx
19	NetrEnumerateTrustedDomains	38	DsrGetDcSiteCoverageW
27	DsrGetDcNameEx	43	DsrGetForestTrustInformation
28	DsrGetSiteName	44	NetrGetForestTrustInformation
33	DsrAddressToSiteNamesW		

Most interesting Netlogon functions that can be called successfully without authentication

Among them, there are actually two unique functions, namely `DsrGetDcNameEx2` and `NetrEnumerateTrustedDomainsEx`, which can be used to obtain all the information provided by the other functions.

Enumerate Domain users and computers

The `DsrGetDcNameEx2` function can be called under OPNUM 34. Microsoft describes it as follows:

> *The **DsrGetDcNameEx2** method return information about a domain controller (DC) in the specified domain and site. If the AccountName parameter is not NULL, and a DC matching the requested capabilities (as defined in the Flags parameter) responds during this method call, then that DC will have verified that the DC account database*

contains an account for the AccountName specified. The server that receives this call is not required to be a DC.

It appears that this function can be used to verify whether or not a user account exists in the domain controller's accounts database.

Let's check which arguments can be passed to this function:

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

It accepts the account name as `AccountName` and a set of bit flags that describe its properties, referred to as `AllowableAccountControlBits`. In addition, it takes as `Flags` the computer name, domain name, domain GUID, site name and another set of bit flags, which provide extra data for processing the request.

According to Microsoft, the server receiving this call doesn't have to be a domain controller (DC). However, in our scenario, we are specifically targeting the domain controller itself, so we can set the other arguments except `AccountName` and `AllowableAccountControlBits` as NULL.

Let's check the AllowableAccountControlBits argument:

Expand table

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
0	0	0	0	0	0	F	0	0	0	0	0	0	0	0	0	0	0	E	D	C	0	B	A	0	0	0	0	0	0	0	0

Where the bits are defined as:

Expand table

Value	Description
A	Account for users whose primary account is in another domain. This account provides user access to the domain, but not to any domain that trusts the domain.
B	Normal domain user account.
C	Interdomain trust account.
D	Computer account for a domain member.
E	Computer account for a backup domain controller (BDC).
F	Computer account for a read-only domain controller (RODC).<148>

All other bits MUST be set to zero and MUST be ignored on receipt.

AllowableAccountControlBits possible values

From the image above, we can see that we can check the account under specific criteria. For our purposes, the most interesting criteria are domain user accounts and computer accounts.

Microsoft states:

> *If the AccountName parameter is not NULL, the response message validation adds the following check: if the DC response is received indicating the lack of an account, as specified in [MS-ADTS] sections 6.3.3 and 6.3.5, the server MUST return ERROR_NO_SUCH_USER.*

So if the account name does not exist, the function should return ERROR_NO_SUCH_USER. In this case, the domain controller will return the error shown below, which is mapped to ERROR_NO_SUCH_USER (The specified account does not exist):

```
▾ Microsoft Network Logon, DsrGetDcNameEx2
    Operation: DsrGetDcNameEx2 (34)
    [Request in frame: 36]
    NULL Pointer: DOMAIN_CONTROLLER_INFO:
    DOS error code: Unknown (0x00000525)
```

Error returned from the domain controller for non-*existent* user

If the user exists, a pointer to a DOMAIN_CONTROLLER_INFOW structure containing data about the DC should be returned, as described by the declaration below:

```
typedef struct _DOMAIN_CONTROLLER_INFOW {
  [string, unique] wchar_t* DomainControllerName;
  [string, unique] wchar_t* DomainControllerAddress;
  ULONG DomainControllerAddressType;
  GUID DomainGuid;
  [string, unique] wchar_t* DomainName;
  [string, unique] wchar_t* DnsForestName;
  ULONG Flags;
  [string, unique] wchar_t* DcSiteName;
  [string, unique] wchar_t* ClientSiteName;
} DOMAIN_CONTROLLER_INFOW,
 *PDOMAIN_CONTROLLER_INFOW;
```

Now let's look at the expected response when providing an existing domain user:

```
DOMAIN_CONTROLLER_INFO:
    DC Name: \\WIN-SO9H290I5NL.test.local
    DC Address: \\192.168.177.177
    DC Address Type: IP/DNS name (1)
    GUID: fc43cec3-957b-464c-b319-3f1ed64bedca
    Logon Domain: test.local
    DNS Forest: test.local
    Domain Controller Flags: 0xe003f3fd
    DC Site: Default-First-Site-Name
    Client Site: Default-First-Site-Name
DOS error code: Success (0x00000000)
```

Information returned from the domain controller for an existing user

From the image above, we can see that the domain controller returns interesting information, including the domain name, domain forest, and domain site.

Another interesting flag inside `AllowableAccountControlBits` is "computer account for a domain member," which is denoted by the value (D). This flag can be used to determine whether the account exists as a computer account in the domain controller database. When a computer is joined to an Active Directory domain, a computer account is created in the directory for that specific computer. This account allows the computer to authenticate to the domain, access domain resources, and participate in domain-wide policies and configurations.

Typically, the computer account will match the host name of the computer (both NetBIOS and DNS). In this case, this flag can be helpful. After collecting the host names of several hosts (e.g., using a tool like nbtscan), we can check whether these hosts belong to the domain or not. The responses returned for existing or non-existent computer accounts are the same as for user accounts.

Before we proceed to the next function, it's worth noting that, as Microsoft mentions, the MS-NRPC interface is also accessible through named pipes, particularly the "netlogon" named pipe, which can be accessed via a null session. Given this, I was curious if there's any public information about enumerating domain users using null sessions. After some searching, I came across some great research by Reino Mostert at Orange Cyber Defense. It discusses enumerating domain users through multiple protocols, including using the `DsrGetDcNameEx2` function over the "netlogon" named pipe. However, it's important to note that this method relies on accessing named pipes, so if null sessions are disabled, this approach won't be effective.

Enumerate trusted domains

The second function we want to discuss is `NetrEnumerateTrustedDomainsEx`, which has OPNUM 36 and is described by Microsoft as follows:

> *The NetrEnumerateTrustedDomainsEx method returns a list of trusted domains from a specified server.*

The function takes only one argument – the server name, which can be NULL:

```
NET_API_STATUS NetrEnumerateTrustedDomainsEx(
  [in, unique, string] LOGONSRV_HANDLE ServerName,
  [out] PNETLOGON_TRUSTED_DOMAIN_ARRAY Domains
);
```

A pointer to a NETLOGON_TRUSTED_DOMAIN_ARRAY structure will be returned that contains an array of DS_DOMAIN_TRUSTSW structures, one for each trusted domain.

This is the declaration of the NETLOGON_TRUSTED_DOMAIN_ARRAY structure:

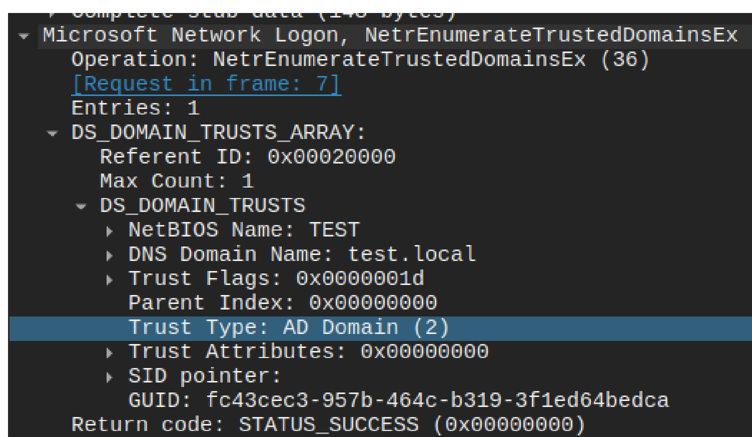
```
typedef struct _NETLOGON_TRUSTED_DOMAIN_ARRAY {
```

```
    DWORD DomainCount;
    [size_is(DomainCount)] PDS_DOMAIN_TRUSTSW Domains;
} NETLOGON_TRUSTED_DOMAIN_ARRAY,
 *PNETLOGON_TRUSTED_DOMAIN_ARRAY;
```

And this is the declaration of the DS_DOMAIN_TRUSTSW structure:

```
typedef struct _DS_DOMAIN_TRUSTSW {
    [string] wchar_t* NetbiosDomainName;
    [string] wchar_t* DnsDomainName;
    ULONG Flags;
    ULONG ParentIndex;
    ULONG TrustType;
    ULONG TrustAttributes;
    PRPC_SID DomainSid;
    GUID DomainGuid;
} DS_DOMAIN_TRUSTSW,
 *PDS_DOMAIN_TRUSTSW;
```

In the image below, you can see the information that is returned by this function:



Information related to trusted domains returned from domain controller

Since there are no trusted domains linked to our test domain controller, it will only return its own domain, as we can see in the screenshot.

Conclusion

In summary, through the No-Auth path, we are able to gather valuable information, including enumerating user and computer accounts, trusted domains, and other information related to the domain. We can also enumerate network interfaces, as was discovered by Airbus. I hope we've been able to return some of the glory of anonymous access after all this time and give you a good strategy for supporting MS-RPC interfaces. You can check out the tool here. But our journey is not yet over. Stay tuned for part two, where we'll dive into MSRPC security and discuss why this type of enumeration remains unstoppable and explore ways to detect it.