Windows Exploitation Tricks: Relaying DCOM Authentication

googleprojectzero.blogspot.com/2021/10/windows-exploitation-tricks-relaying.html

Posted by James Forshaw, Project Zero

In my previous blog post I discussed the possibility of relaying Kerberos authentication from a DCOM connection. I was originally going to provide a more in-depth explanation of how that works, but as it's quite involved I thought it was worthy of its own blog post. This is primarily a technique to get relay authentication from another user on the same machine and forward that to a network service such as LDAP. You could use this to escalate privileges on a host using a technique similar to a blog post from Shenanigans Labs but removing the requirement for the WebDAV service. Let's get straight to it.

Background

The technique to locally relay authentication for DCOM was something I originally reported back in 2015 (issue 325). This issue was fixed as CVE-2015-2370, however the underlying authentication relay using DCOM remained. This was repurposed and expanded upon by various others for local and remote privilege escalation in the RottenPotato series of exploits, the latest in that line being RemotePotato which is currently unpatched as of October 2021.

The key feature that the exploit abused is standard COM marshaling. Specifically when a COM object is marshaled so that it can be used by a different process or host, the COM runtime generates an OBJREF structure, most commonly the OBJREF_STANDARD form. This structure contains all the information necessary to establish a connection between a COM client and the original object in the COM server.

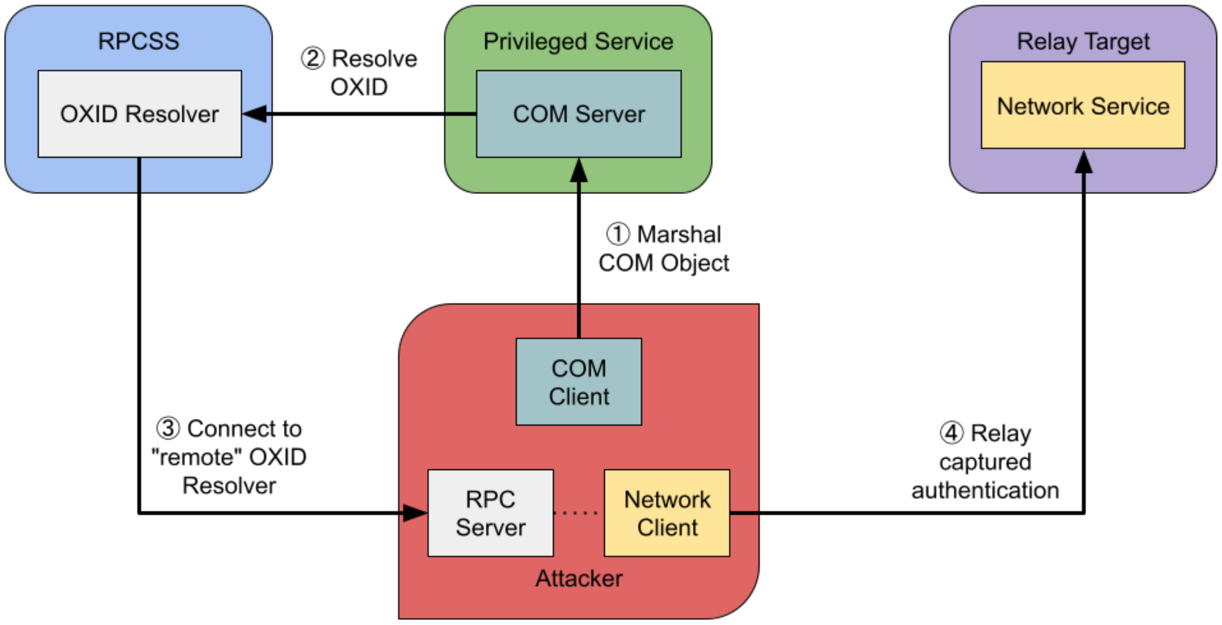
Connecting to the original object from the OBJREF is a two part process:

1. The client extracts the Object Exporter ID (OXID) from the structure and contacts the OXID resolver service specified by the RPC binding information in the OBJREF.
2. The client uses the OXID resolver service to find the RPC binding information of the COM server which hosts the object and establishes a connection to the RPC endpoint to access the object's interfaces.

Both of these steps require establishing an MSRPC connection to an endpoint. Commonly this is either locally over ALPC, or remotely via TCP. If a TCP connection is used then the client will also authenticate to the RPC server using NTLM or Kerberos based on the security bindings in the OBJREF.

The first key insight I had for issue 325 is that you can construct an OBJREF which will always establish a connection to the OXID resolver service over TCP, even if the service was on the local machine. To do this you specify the hostname as an IP address and an arbitrary TCP port for the client to connect to. This allows you to listen locally and when the RPC connection is made the authentication can be relayed or repurposed.

This isn't yet a privilege escalation, since you need to convince a privileged user to unmarshal the OBJREF. This was the second key insight: you could get a privileged service to unmarshal an arbitrary OBJREF easily using the CoGetInstanceFromIStorage API and activating a privileged COM service. This marshals a COM object, creates the privileged COM server and then unmarshals the object in the server's security context. This results in an RPC call to the fake OXID resolver authenticated using a privileged user's credentials. From there the authentication could be relayed to the local system for privilege escalation.



Being able to redirect the OXID resolver RPC connection locally to a different TCP port was not by design and Microsoft eventually fixed this in Windows 10 1809/Server 2019. The underlying issue prior to Windows 10 1809 was the string containing the host returned as part of the OBJREF was directly concatenated into an RPC string binding. Normally the RPC string binding should have been in the form of:

```
ncacn_ip_tcp:ADDRESS[135]
```

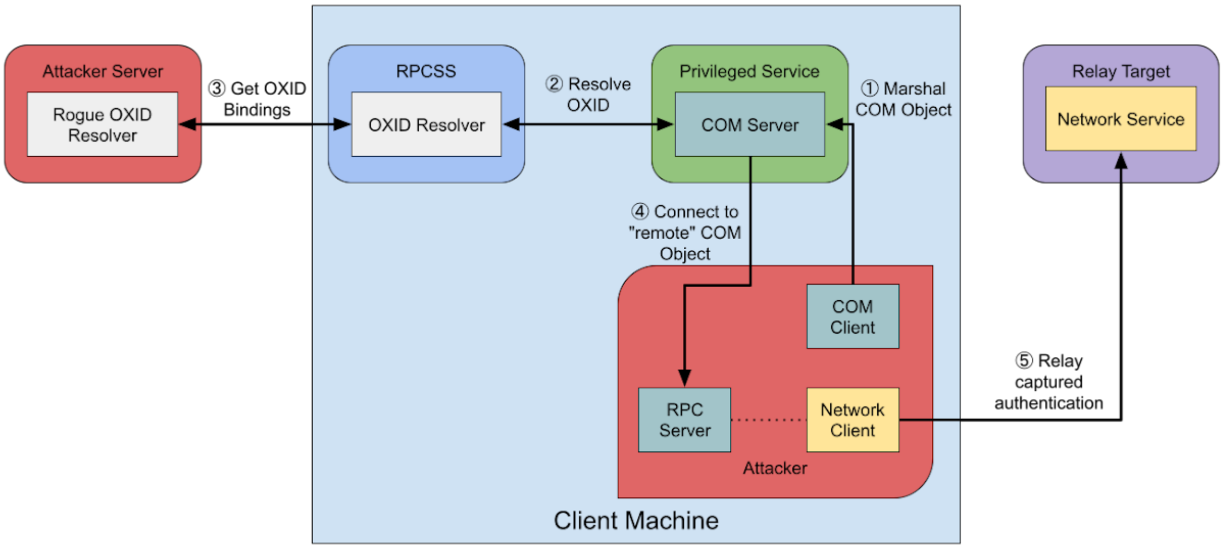
Where ncacn_ip_tcp is the protocol sequence for RPC over TCP, ADDRESS is the target address which would come from the string binding, and [135] is the well-known TCP port for the OXID resolver appended by RPCSS. However, as the ADDRESS value is inserted manually into the binding then the OBJREF could specify its own port, resulting in the string binding:

```
ncacn_ip_tcp:ADDRESS[9999][135]
```

The RPC runtime would just pick the first port in the binding string to connect to, in this case 9999, and would ignore the second port 135. This behavior was fixed by calling the RpcStringBindingCompose API which will correctly escape the additional port number which ensures it's ignored when making the RPC connection.

This is where the RemotePotato exploit, developed by Antonio Cocomazzi and Andrea Pierini, comes into the picture. While it was no longer possible to redirect the OXID resolving to a local TCP server, you could redirect the initial connection to an external server. A call is made to the IObjectExporter::ResolveOxid2 method which can return an arbitrary RPC binding string for a fake COM object.

Unlike the OXID resolver binding string, the one for the COM object is allowed to contain an arbitrary TCP port. By returning a binding string for the original host on an arbitrary TCP port, the second part of the connection process can be relayed rather than the first. The relayed authentication can then be sent to a domain server, such as LDAP or SMB, as long as they don't enforce signing.



This exploit has the clear disadvantage of requiring an external machine to act as the target of the initial OXID resolving. While investigating the Kerberos authentication relay attacks for DCOM, could I find a way to do everything on the same machine?

Remote ➜ Local Potato

If we're relaying the authentication for the second RPC connection, could we get the local OXID resolver to do the work for us and resolve to a local COM server on a randomly selected port? One of my goals is to write the least amount of code, which is why we'll do everything in C# and .NET.

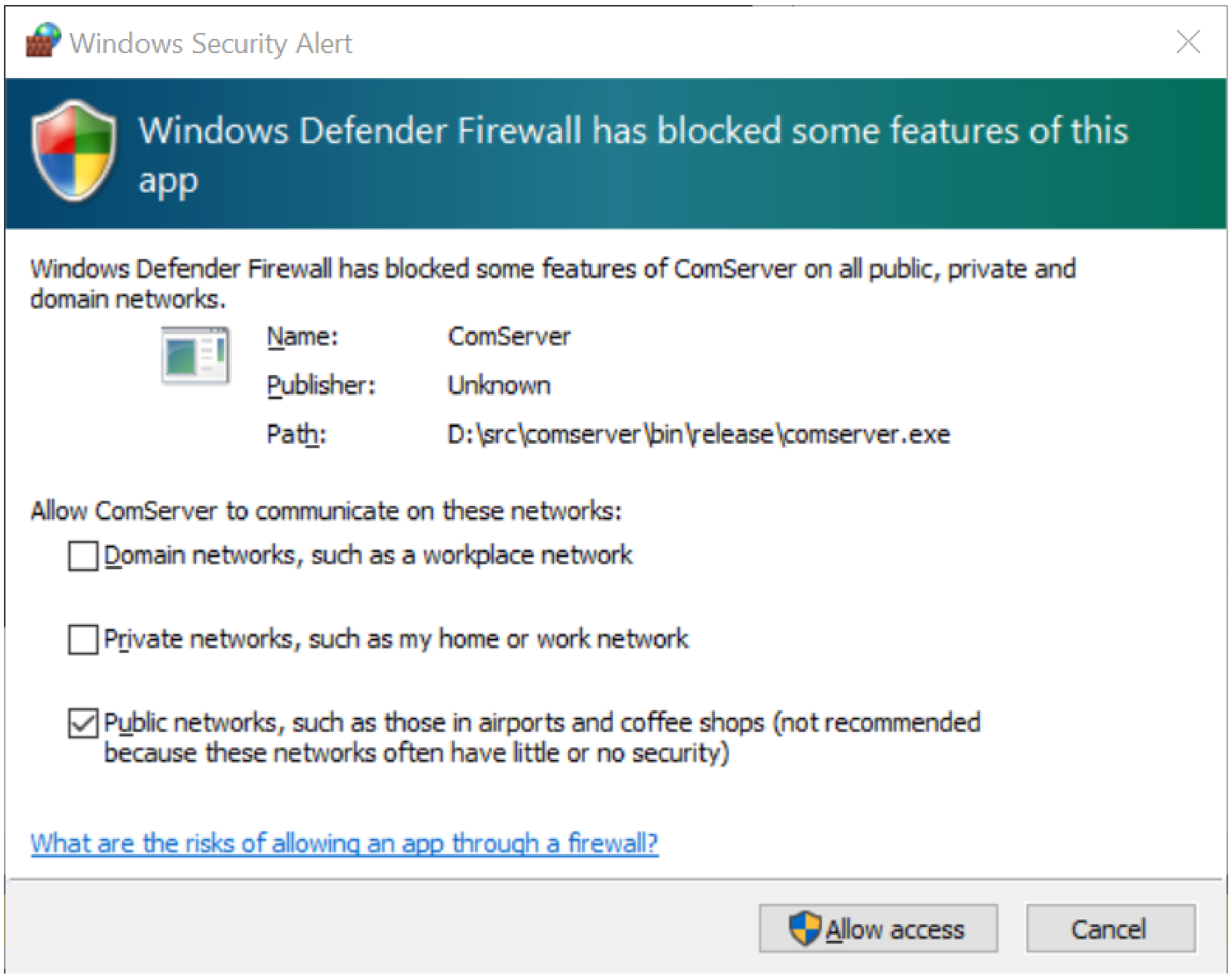
```
byte[] ba = GetMarshalledObject(new object());
var std = COMObjRefStandard.FromArray(ba);
Console.WriteLine("IPID: {0}", std.Ipid);
Console.WriteLine("OXID: {0:X08}", std.Oxid);
Console.WriteLine("OID : {0:X08}", std.Oid);
std.StringBindings.Clear();
std.StringBindings.Add(RpcTowerId.Tcp, "127.0.0.1");
Console.WriteLine($"objref:{0}:", Convert.ToBase64String(std.ToArray());
```

This code creates a basic .NET object and COM marshals it to a standard OBJREF. I've left out the code for the marshalling and parsing of the OBJREF, but much of that is already present in the linked issue 325. We then modify the list of string bindings to only include a TCP binding for 127.0.0.1, forcing the OXID resolver to use TCP. If you specify a computer's hostname then the OXID resolver will use ALPC instead. Note that the string bindings in the OBJREF are only for binding to the OXID resolver, not the COM server itself.

We can then convert the modified OBJREF into an objref moniker. This format is useful as it allows us to trivially unmarshal the object in another process by calling the Marshal::BindToMoniker API in .NET and passing the moniker string. For example to bind to the COM object in PowerShell you can run the following command:

```
[Runtime.InteropServices.Marshal]::BindToMoniker("objref:TUVP...:")
```

Immediately after binding to the moniker a firewall dialog is likely to appear as shown:


Windows Security Alert

Windows Defender Firewall has blocked some features of this app

Windows Defender Firewall has blocked some features of ComServer on all public, private and domain networks.

Name:	ComServer
Publisher:	Unknown
Path:	D:\src\comserver\bin\release\comserver.exe

Allow ComServer to communicate on these networks:

- [ ] Domain networks, such as a workplace network
- [ ] Private networks, such as my home or work network
- [✓] Public networks, such as those in airports and coffee shops (not recommended because these networks often have little or no security)

What are the risks of allowing an app through a firewall?

This is requesting the user to allow our COM server process access to listen on all network interfaces for incoming connections. This prompt only appears when the client tries to resolve the OXID as DCOM supports dynamic RPC endpoints. Initially when the COM server starts it only listens on ALPC, but the RPCSS service can ask the server to bind to additional endpoints.

This request is made through an internal RPC interface that every COM server implements for use by the RPCSS service. One of the functions on this interface is UseProtSeq, which requests that the COM server enables a TCP endpoint. When the COM server receives the UseProtSeq call it tries to bind a TCP server to all interfaces, which subsequently triggers the Windows Defender Firewall to prompt the user for access.

Enabling the firewall permission requires administrator privileges. However, as we only need to listen for connections via localhost we shouldn't need to modify the firewall so the dialog can be dismissed safely. However, going back to the COM client we'll see an error reported.

```
Exception calling "BindToMoniker" with "1" argument(s):

"The RPC server is unavailable. (Exception from HRESULT: 0x800706BA)"
```

If we allow our COM server executable through the firewall, the client is able to connect over TCP successfully. Clearly the firewall is affecting the behavior of the COM client in some way even though it shouldn't. Tracing through the unmarshalling process in the COM client, the error is being returned from RPCSS when trying to resolve the OXID's binding information. This would imply that no connection attempt is made, and RPCSS is detecting that the COM server wouldn't be allowed through the firewall and refusing to return any binding information for TCP.

Further digging into RPCSS led me to the following function:

```
BOOL IsPortOpen(LPWSTR ImageFileName, int PortNumber) {
  INetFwMgr* mgr;


  CoCreateInstance(CLSID_FwMgr, NULL, CLSCTX_INPROC_SERVER,
                   IID_PPV_ARGS(&mgr));
  VARIANT Allowed;
  VARIANT Restricted;
  mgr->IsPortAllowed(ImageFileName, NET_FW_IP_VERSION_ANY,
           PortNumber, NULL, NET_FW_IP_PROTOCOL_TCP,
           &Allowed, &Restricted);
  if (VT_BOOL != Allowed.vt)
    return FALSE;
  return Allowed.boolVal == VARIANT_TRUE;
}
```

This function uses the HNetCfg.FwMgr COM object, and calls INetFwMgr::IsPortAllowed to determine if the process is allowed to listen on the specified TCP port. This function is called for every TCP binding when enumerating the COM server's bindings to return to the client. RPCSS passes the full path to the COM server's executable and the listening TCP port. If the function returns FALSE then RPCSS doesn't consider it valid and won't add it to the list of potential bindings.

If the OXID resolving process doesn't have any binding at the end of the lookup process it will return the RPC_S_SERVER_UNAVAILABLE error and the COM client will fail to bind to the server. How can we get around this limitation without needing administrator privileges to allow our server through the firewall? We can convert this C++ code into a small PowerShell function to test the behavior of the function to see what would grant access.

```
function Test-IsPortOpen {
    param(
        [string]$Name,
        [int]$Port
    )
    $mgr = New-Object -ComObject "HNetCfg.FwMgr"
    $allow = $null
    $mgr.IsPortAllowed($Name, 2, $Port, "", 6, [ref]$allow, $null)
    $allow
}
foreach($f in $(ls "$env:WINDIR\system32\*.exe")) {
    if (Test-IsPortOpen $f.FullName 12345) {
        Write-Host $f.Fullname
    }
}
```

This script enumerates all executable files in system32 and checks if they'd be allowed to connect to TCP port 12345. Normally the TCP port would be selected automatically, however the COM server can use the RpcServerUseProtseqEp API to pre-register a known TCP port for RPC communication, so we'll just pick port 12345.

The only executable in system32 that returns true from Test-IsPortOpen is svchost.exe. That makes some sense, the default firewall rules usually permit a limited number of services to be accessible through the firewall, the majority of which are hosted in a shared service process.

This check doesn't guarantee a COM server will be allowed through the firewall, just that it's potentially accessible in order to return a TCP binding string. As the connection will be via localhost we don't need to be allowed through the firewall, only that IsPortOpen thinks we could be open. How can we spoof the image filename?

The obvious trick would be to create a svchost.exe process and inject our own code in there. However, that is harder to achieve through pure .NET code and also injecting into an svchost executable is a bit of a red flag if something is monitoring for malicious code which might make the exploit unreliable. Instead, perhaps we can influence the image filename used by RPCSS?

Digging into the COM runtime, when a COM server registers itself with RPCSS it passes its own image filename as part of the registration information. The runtime gets the image filename through a call to GetModuleFileName, which gets the value from the ImagePathName field in the process parameters block referenced by the PEB.

We can modify this string in our own process to be anything we like, then when COM is initialized, that will be sent to RPCSS which will use it for the firewall check. Once the check passes, RPCSS will return the TCP string bindings for our COM server when unmarshalling the OBJREF and the client will be able to connect. This can all be done with only minor in-process modifications from .NET and no external servers required.

Capturing Authentication

At this point a new RPC connection will be made to our process to communicate with the marshaled COM object. During that process the COM client must authenticate, so we can capture and relay that authentication to another service locally or remotely. What's the best way to capture that authentication traffic?

Before we do anything we need to select what authentication we want to receive, and this will be reflected in the OBJREF's security bindings. As we're doing everything using the existing COM runtime we can register what RPC authentication services to use when calling CoInitializeSecurity in the COM server through the asAuthSvc parameter.

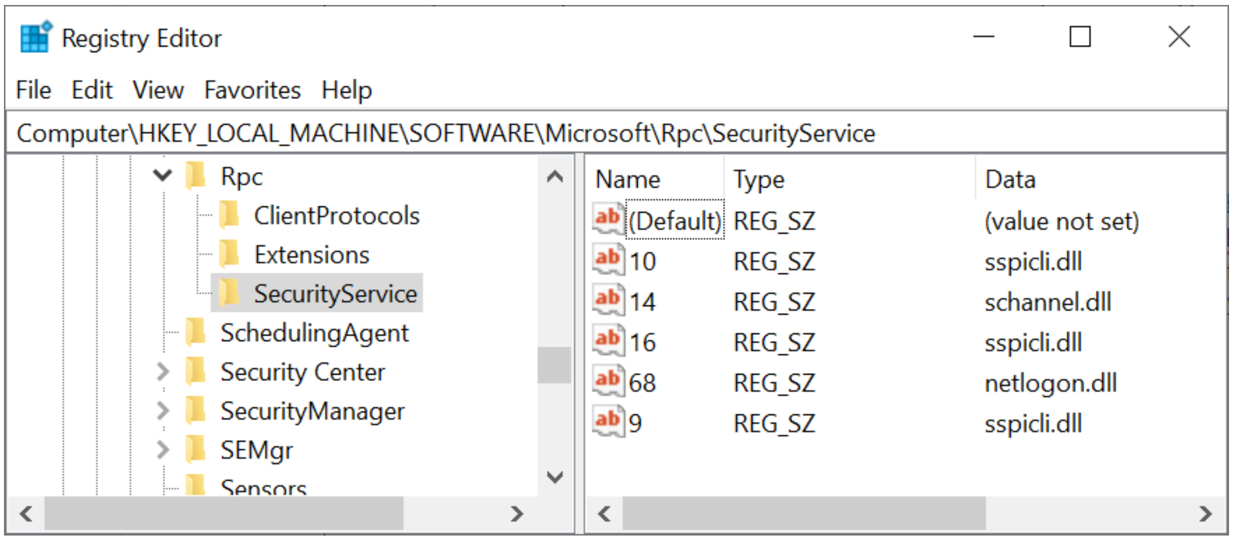
```
var svcs = new SOLE_AUTHENTICATION_SERVICE[] {
    new SOLE_AUTHENTICATION_SERVICE() {
      dwAuthnSvc = RpcAuthenticationType.Kerberos,
      pPrincipalName = "HOST/DC.domain.com"
    }
};
var str = SetProcessModuleName("System");
try
{
   CoInitializeSecurity(IntPtr.Zero, svcs.Length, svcs,
        IntPtr.Zero, AuthnLevel.RPC_C_AUTHN_LEVEL_DEFAULT,
        ImpLevel.RPC_C_IMP_LEVEL_IMPERSONATE, IntPtr.Zero,
        EOLE_AUTHENTICATION_CAPABILITIES.EOAC_DYNAMIC_CLOAKING,
        IntPtr.Zero);
}
finally
{
    SetProcessModuleName(str);
}
```

In the above code, we register to only receive Kerberos authentication and we can also specify an arbitrary SPN as I described in the previous blog post. One thing to note is that the call to CoInitializeSecurity will establish the connection to RPCSS and pass the executable filename. Therefore we need to modify the filename before calling the API as we can't change it after the connection has been established.

For swag points I specify the filename System rather than build the full path to svchost.exe. This is the name assigned to the kernel which is also granted access through the firewall. We restore the original filename after the call to CoInitializeSecurity to reduce the risk of it breaking something unexpectedly.

That covers the selection of the authentication service to use, but doesn't help us actually capture that authentication. My first thought to capture the authentication was to find the socket handle for the TCP server, close it and create a new socket in its place. Then I could directly process the RPC protocol and parse out the authentication. This felt somewhat risky as the RPC runtime would still think it has a valid TCP server socket and might fail in unexpected ways. Also it felt like a lot of work, when I have a perfectly good RPC protocol parser built into Windows.

I then resigned myself to hooking the SSPI APIs, although ideally I'd prefer not to do so. However, once I started looking at the RPC runtime library there weren't any imports for the SSPI APIs to hook into and I really didn't want to patch the functions themselves. It turns out that the RPC runtime loads security packages dynamically, based on the authentication service requested and the configuration of the HKLM\SOFTWARE\Microsoft\Rpc\SecurityService registry key.



Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\SecurityService

Name	Type	Data
(Default)	REG_SZ	(value not set)
10	REG_SZ	sspicli.dll
14	REG_SZ	schannel.dll
16	REG_SZ	sspicli.dll
68	REG_SZ	netlogon.dll
9	REG_SZ	sspicli.dll

The key, shown in the above screenshot has a list of values. The value's name is the number assigned to the authentication service, for example 16 is RPC_C_AUTHN_GSS_KERBEROS. The value's data is then the name of the DLL to load which provides the API, for Kerberos this is sspicli.dll.

The RPC runtime then loads a table of security functions from the DLL by calling its exported InitSecurityInterface method. At least for sspicli the table is always the same and is a pre-initialized structure in the DLL's data section. This is perfect, we can just call InitSecurityInterface before the RPC runtime is initialized to get a pointer to the table then modify its function pointers to point to our own implementation of the API. As an added bonus the table is in a writable section of the DLL so we don't even need to modify the memory protection.

Of course implementing the hooks is non-trivial. This is made more complex because RPC uses the DCE style Kerberos authentication which requires two tokens from the client before the server considers the authentication complete. This requires maintaining more state to keep the RPC server and client implementations happy. I'll leave this as an exercise for the reader.

Choosing a Relay Target Service

The next step is to choose a suitable target service to relay the authentication to. For issue 325 I relayed the authentication to the same machine's DCOM activator RPC service and was able to achieve an arbitrary file write.

I thought that maybe I could do so again, so I modified my .NET RPC client to handle the relayed authentication and tried accessing local RPC services. No matter what RPC server or function I called, I always got an access denied error. Even if I wrote my own RPC server which didn't have any checks, it would fail.

Digging into the failure it turned out that at some point (I don't know specifically when), Microsoft added a mitigation into the RPC runtime to make it very difficult to relay authentication back to the same system.

```
void SSECURITY_CONTEXT::ValidateUpgradeCriteria() {
  if (this->AuthnLevel < RPC_C_AUTHN_LEVEL_PKT_INTEGRITY) {
    if (IsLoopback())
      this->UnsafeLoopbackAuth = TRUE;
  }
}
```

The SSECURITY_CONTEXT::ValidateUpgradeCriteria method is called when receiving RPC authentication packets. If the authentication level for the RPC connection is less than RPC_C_AUTHN_LEVEL_PKT_INTEGRITY such as RPC_C_AUTHN_LEVEL_PKT_CONNECT and the authentication was from the same system then a flag is set to true in the security context. The IsLoopback function calls the QueryContextAttributes API for the undocumented SECPKG_ATTR_IS_LOOPBACK attribute value from the server security context. This attribute indicates if the authentication was from the local system.

When an RPC call is made the server will check if the flag is true, if it is then the call will be immediately rejected before any code is called in the server including the RPC interface's security callback. The only way to pass this check is either the authentication doesn't come from the local system or the authentication level is RPC_C_AUTHN_LEVEL_PKT_INTEGRITY or above which then requires the client to know the session key for signing or encryption. The RPC client will also check for local authentication and will increase the authentication level if necessary. This is an effective way of preventing the relay of local authentication to elevate privileges.

Instead as I was focussing on Kerberos, I came to the conclusion that relaying the authentication to an enterprise network service was more useful. As the default settings for a domain controller's LDAP service still do not enforce signing, it would seem a reasonable target. As we'll see, this provides a limitation of the source of the authentication, as it must not enable Integrity otherwise the LDAP server will enforce signing.

The problem with LDAP is I didn't have any code which implemented the protocol. I'm sure there is some .NET code to do it somewhere, but the fewer dependencies I have the better. As I mentioned in the previous blog post, Windows has a builtin LDAP library in wldap32.dll. Could I repurpose its API but convert it into using relayed authentication?

Unsurprisingly the library doesn't have a "Enable relayed authentication" mode, but after a few minutes in a disassembler, it was clear it was also delay-loading the SSPI interfaces through the InitSecurityInterface method. I could repurpose my code for capturing the authentication for relaying the authentication. There was initially a minor issue, accidentally or on purpose there was a stray call to QueryContextAttributes which was directly imported, so I needed to patch that through an Import Address Table (IAT) hook as distasteful as that was.

There was still a problem however. When the client connects it always tries to enable LDAP signing, as we are relaying authentication with no access to the session key this causes the connection to fail. Setting the option value LDAP_OPT_SIGN in the library to false didn't change this behavior. I needed to set the LdapClientIntegrity registry value to 0 in the LDAP service's key before initializing the library. Unfortunately that key is only modifiable by administrators. I could have modified the library itself, but as it was checking the key during DllMain it would be a complex dance to patch the DLL in the middle of loading.

Instead I decided to override the HKEY_LOCAL_MACHINE key. This is possible for the Win32 APIs by using the RegOverridePredefKey API. The purpose of the API is to allow installers to redirect administrator-only modifications to the registry into a writable location, however for our purposes we can also use it to redirect the reading of the LdapClientIntegrity registry value.

```
[DllImport("Advapi32.dll")]
static extern int RegOverridePredefKey(
    IntPtr hKey,
    IntPtr hNewHKey
);
[DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
static extern IntPtr LoadLibrary(string libname);
static readonly IntPtr HKEY_LOCAL_MACHINE = new IntPtr(-2147483646);
static void OverrideLocalMachine(RegistryKey key)
{
    int res = RegOverridePredefKey(HKEY_LOCAL_MACHINE,
        key?.Handle.DangerousGetHandle() ?? IntPtr.Zero);
    if (res != 0)
        throw new Win32Exception(res);
}
static void LoadLDAPLibrary()
{
    string dummy = @"SOFTWARE\DUMMY";
    string target = @"System\CurrentControlSet\Services\LDAP";
    using (var key = Registry.CurrentUser.CreateSubKey(dummy, true))
    {
        using (var okey = key.CreateSubKey(target, true))
        {
            okey.SetValue("LdapClientIntegrity", 0,
                          RegistryValueKind.DWord);
            OverrideLocalMachine(key);
            try
            {
                IntPtr lib = LoadLibrary("wldap32.dll");
                if (lib == IntPtr.Zero)
                    throw new Win32Exception();
            }
            finally
            {
                OverrideLocalMachine();
                Registry.CurrentUser.DeleteSubKeyTree(dummy);
            }
        }
    }
}
```

This code redirects the HKEY_LOCAL_MACHINE key and then loads the LDAP library. Once it's loaded we can then revert the override so that everything else works as expected. We can now repurpose the built-in LDAP library to relay Kerberos authentication to the domain controller. For the final step, we need a privileged COM service to unmarshal the OBJREF to start the process.

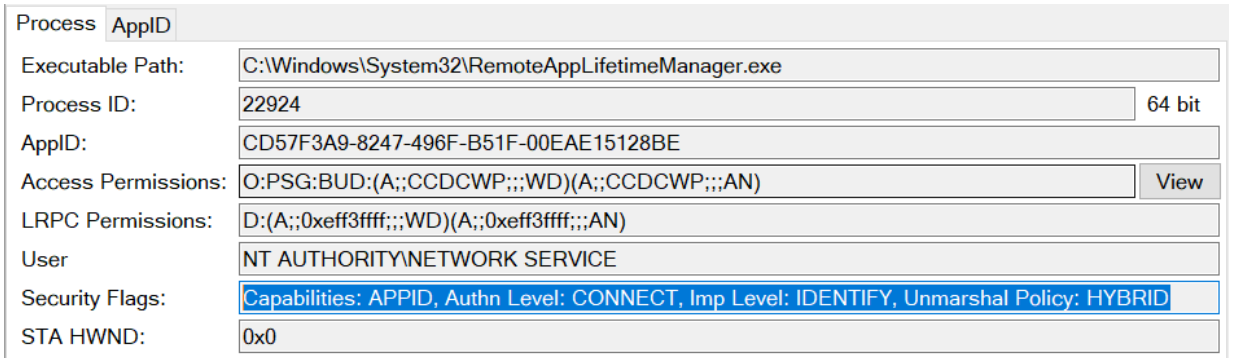
Choosing a COM Unmarshaller

The RemotePotato attack assumes that a more privileged user is authenticated on the same machine. However I wanted to see what I could do without that requirement. Realistically the only thing that can be done is to relay the computer's domain account to the LDAP server.

To get access to authentication for the computer account, we need to unmarshal the OBJREF inside a process running as either SYSTEM or NETWORK SERVICE. These local accounts are mapped to the computer account when authenticating to another machine on the network.

We do have one big limitation on the selection of a suitable COM server: it must make the RPC connection using the RPC_C_AUTHN_LEVEL_PKT_CONNECT authentication level. Anything above that will enable Integrity on the authentication which will prevent us relaying to LDAP. Fortunately RPC_C_AUTHN_LEVEL_PKT_CONNECT is the default setting for DCOM, but unfortunately all services which use the svchost process change that default to RPC_C_AUTHN_LEVEL_PKT which enables Integrity.

After a bit of hunting around with OleViewDotNet, I found a good candidate class, CRemoteAppLifetimeManager (CLSID: 0bae55fc-479f-45c2-972e-e951be72c0c1) which is hosted in its own executable, runs as NETWORK SERVICE, and doesn't change any default settings as shown below.



Executable Path:	C:\Windows\System32\RemoteAppLifetimeManager.exe
Process ID:	22924 64 bit
AppID:	CD57F3A9-8247-496F-B51F-00EAE15128BE
Access Permissions:	O:PSG:BUD:(A;;CCDCWP;;;WD)(A;;CCDCWP;;;AN)
LRPC Permissions:	D:(A;;0xeff3ffff;;;WD)(A;;0xeff3ffff;;;AN)
User	NT AUTHORITY\NETWORK SERVICE
Security Flags:	Capabilities: APPID, Authn Level: CONNECT, Imp Level: IDENTIFY, Unmarshal Policy: HYBRID
STA HWND:	0x0

The server doesn't change the default impersonation level from RPC_C_IMP_LEVEL_IDENTIFY, which means the negotiated token will only be at SecurityIdentification level. For LDAP, this doesn't matter as it only uses the token for access checking, not to open resources. However, this would prevent using the same authentication to access something like the SMB server. I'm confident that given enough effort, a COM server with both RPC_C_AUTHN_LEVEL_PKT_CONNECT and RPC_C_IMP_LEVEL_IMPERSONATE could be found, but it wasn't necessary for my exploit.

Wrapping Up

That's a somewhat complex exploit. However, it does allow for authentication relay, with arbitrary Kerberos tokens from a local user to LDAP on a default Windows 10 system. Hopefully it might provide some ideas of how to implement something similar without always needing to write your protocol servers and clients and just use what's already available.

This exploit is very similar to the existing RemotePotato exploit that Microsoft have already stated will not be fixed. This is because Microsoft considers authentication relay attacks to be an issue with the configuration of the Windows network, such as not enforcing signing on LDAP, rather than the particular technique used to generate the authentication relay. As I mentioned in the previous blog post, at most this would be assessed as a Moderate severity issue which does not reach the bar for fixing as part of regular updates (or potentially, not being fixed at all).

As for mitigating this issue without it being fixed by Microsoft, a system administrator should follow Microsoft's recommendations to enable signing and/or encryption on any sensitive service in the domain, especially LDAP. They can also enable Extended Protection for Authentication where the service is protected by TLS. They can also configure the default DCOM authentication level to be RPC_C_AUTHN_LEVEL_PKT_INTEGRITY or above. These changes would make the relay of Kerberos, or NTLM significantly less useful.