Bypassing UAC with SSPI Datagram Contexts

splintercod3.blogspot.com/p/bypassing-uac-with-sspi-datagram.html

splintercode

by splinter_code - 14 September 2023

Recently i had the opportunity to read through some of my old repos because i wanted to reuse some code i used for a UIPI bypass in the past, aiming to adapt it to a new hidden feature of the task manager for a sneaky and for-fun UAC bypass.

Luckily, i stumbled upon another UAC related project (a 2 years old project) in which i tried to implement an idea to bypass UAC through some particular SSPI configurations, but i failed miserably that time.

Upon re-reading the code, a light bulb came to my mind so i tried a different exploitation path and it ended up with a new cool **UAC bypass**! So let's jump straight to it 👇

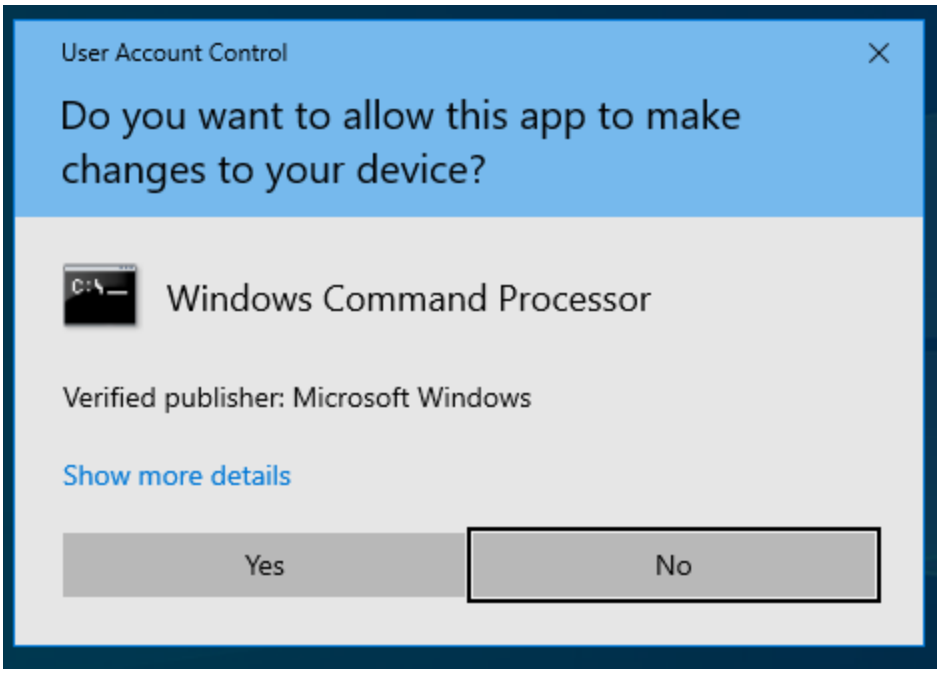
UAC: User Account Control (formerly LUA - Limited User Account)

To provide some context, UAC (User Account Control) is an elevation mechanism within Windows designed to trigger a **consent** prompt when an action requires administrative privileges. This consent prompt is intended to enforce **privilege separation** by requiring administrator approval.

While designed to add an extra layer of security against unauthorized OS changes, it has been proven to be a full of holes design.
There are many **known** ways to bypass UAC and perform actions with elevated privileges without any prompt or consent provided interactively by the user.

You can consult UACMe for a curated list and related source code of known UAC bypasses (fixed and unfixed 🐵).

I bet you have encountered this screen at some point. Yep, that's the UAC consent prompt for you:



In this post i'm not going to detail the internal working of UAC, but if you are interested in knowing more there is already a lot of research about it. You can find some comprehensive talks and blog posts in the References section.

An interesting behavior in NTLM authentications

In Windows exists the fantastic concept of “type your password once and authenticate everywhere”. This is the same basic concept as any Single Sign-On system but integrated directly into the operating system.

In order for this to work someone has to store your passwords and that’s where the **LSA** comes into play. It provides a layer of abstraction for any related authentication happening on your system.

Without going into the deep details, what you need to know is that the LSA (implemented in **lsass.exe**) loads **authentication packages** DLLs by using configuration information stored in the registry. Loading multiple authentication packages permits the LSA to support multiple security protocols, e.g. NTLM, Kerberos and so on…

When you log on interactively, the LSA creates a new **logon session**, associates it with your credentials, and creates a **token** for your process that references this newly created logon session.
In this way, when your process tries to access a remote resource, let’s say \\SHARE-SERVER\share1\file.txt, your process can invoke the SSPI functions to retrieve the **security buffers** to send over the network wire and the authentication is abstracted from the application logic without the needs of providing explicit credentials.
What happens under the hood is that when your application invokes the SSPI functions, it communicates with lsass.exe, which in turn will inspect your process (or thread if impersonating) token and will be able to associate your right **credentials** and derive the proper authentication buffers that your process can use for the authentication.

This is an oversimplified explanation, but hopefully you got the point.

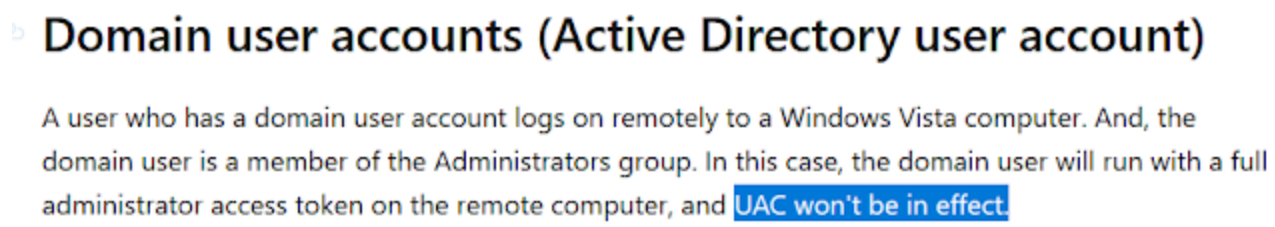
When **network authentication** takes place, UAC restrictions don't affect the generated token.
There are 2 exceptions to this rule:

- if you're authenticating to a remote machine using a **shared local administrator** account (except built-in Administrator);

- if you are doing a **loopback authentication** without SPPI and using a local administrator user. You need to know the password or at least the hash of the user.

Only in these cases UAC Remote restrictions kick in.
These restrictions will limit also the token generated by the network authentication on the server end if LocalAccountTokenFilterPolicy is set to 0, which is the default configuration.

Instead, if you use a domain user which is also an administrator of the machine, UAC won't get in the way:



UAC Remote restrictions for domain users

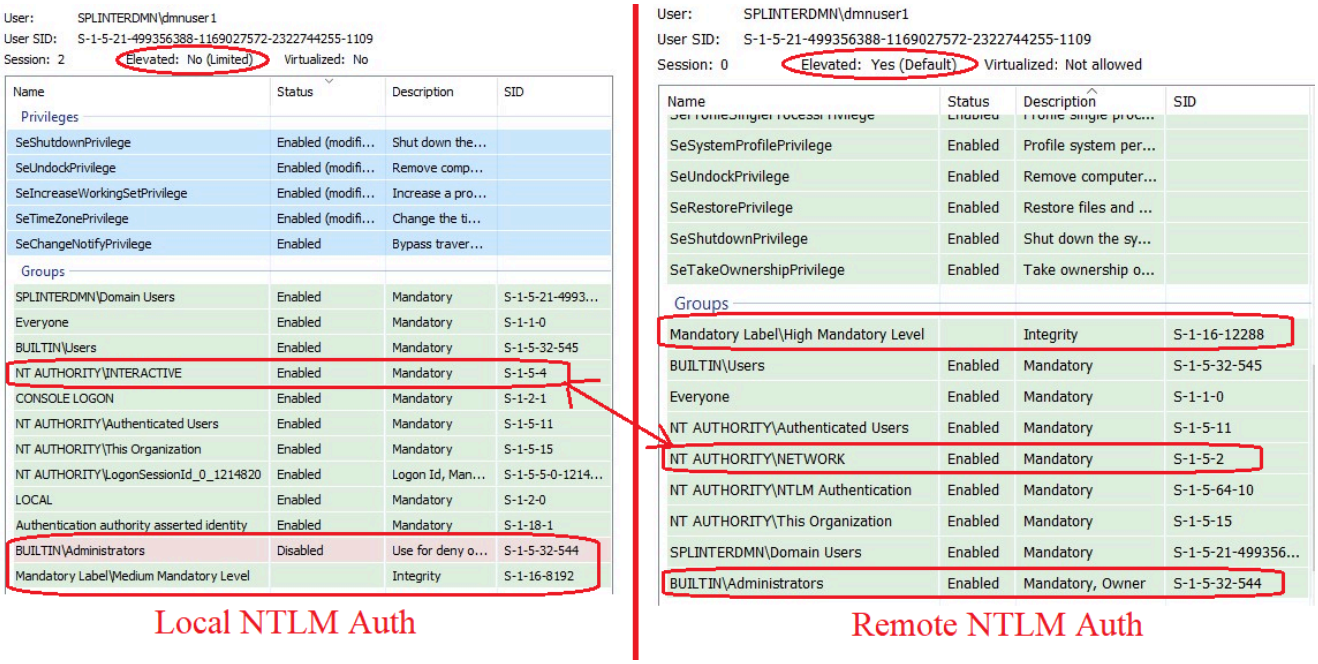
The main mechanism that is preventing anyone from getting around UAC locally through SSPI is Local Authentication.
To understand it, let's take out of the equation the local authentication with Kerberos and focus on **NTLM**. (NOTE: James Forshaw already demonstrated how UAC restrictions over Kerberos can be bypassed locally in this blogpost)

If you are familiar with NTLM authentications, you can identify a local authentication by observing these details in the messages exchange:

- The server sets the "**Negotiate Local Call**" flag in the Challenge message (Type 2);
- The "**Reserved**" field in the Challenge message is not 0 and contains a number referencing the server context handle;
- The generated Authenticate message (Type 3) by the client contains **empty** security buffers;

When this occurs, LSASS is able to associate the calling process's **actual token** with the server application's security context. As a result, any UAC restrictions on the client side become visible to the server application.

Ok, enough theory. Let's see the differences in the token when doing a local vs remote NTLM authentication through SSPI:



System Informer token views of local auth (left) vs remote auth (right)

The result is that the token returned from the local authentication has the UAC limitations, in fact you can see the **IL level** is **Medium** and the **Administrators** SID is **disabled**.
Instead, the remote authentication occurred without UAC limitations and the resulting elevated token is set in **High IL**.
One important difference here is in the logon type SID present in the token, on the filtered token there is the INTERACTIVE SID while in the elevated token there is the NETWORK SID.

So the 1 million $ question is: can we fake a network authentication locally with NTLM through SSPI?

The unexpected bit flag

If we want to trick LSASS during local authentications, first we need to understand when and how this decision takes place in the code.
Let's reverse msv1_0.dll and search for the function which sets the flag 0x4000 (NTLMSSP_NEGOTIATE_LOCAL_CALL):

```
else
{
  oemDomainName.Buffer = (PCHAR)&NegotiateMessage->Signature[oemDomainNameBufferIndex];
  oemDomainName.Length = NegotiateMessage->OemDomainName.Length;
  oemDomainName.MaximumLength = oemDomainName.Length;
  oemWorkstationNameBufferIndex = NegotiateMessage->OemWorkstationName.Buffer;
  if ( oemWorkstationNameBufferIndex < inputTokenSize
    && oemWorkstationNameBufferIndex + NegotiateMessage->OemWorkstationName.Length <= inputTokenSize )
  {
    oemWorkstationNameBuffer.Buffer = (PCHAR)&NegotiateMessage->Signature[oemWorkstationNameBufferIndex];
    oemWorkstationNameBuffer.Length = NegotiateMessage->OemWorkstationName.Length;
    oemWorkstationNameBuffer.MaximumLength = oemWorkstationNameBuffer.Length;
    RtlAcquireResourceShared(&NtLmGlobalCritSect, TRUE);
    if ( RtlEqualString(&oemWorkstationNameBuffer, &NtLmGlobalOemPhysicalComputerNameString, 0) )
    {
      if ( RtlEqualString(&oemDomainName, &NtLmGlobalOemPrimaryDomainNameString, 0) )
      {
        challengeMessage->NegotiateFlags |= NTLMSSP_NEGOTIATE_LOCAL_CALL;
```

SsprHandleNegotiateMessage reversed code that sets the “Negotiate Local Call” flag

Without surprise we landed to the function SsprHandleNegotiateMessage. What this function does is to handle the Negotiate message received by the client and generate the proper Challenge. From the code perspective we land here in the first server call to AcceptSecurityContext.
The logic of this code for detecting a local authentication is pretty straightforward: if the domain name and machine name provided by the client in the **Negotiate** message matches with the local machine name and domain, then this is a local authentication case.

But how we get into this part in the code? Let’s cross reference the if above that branch:

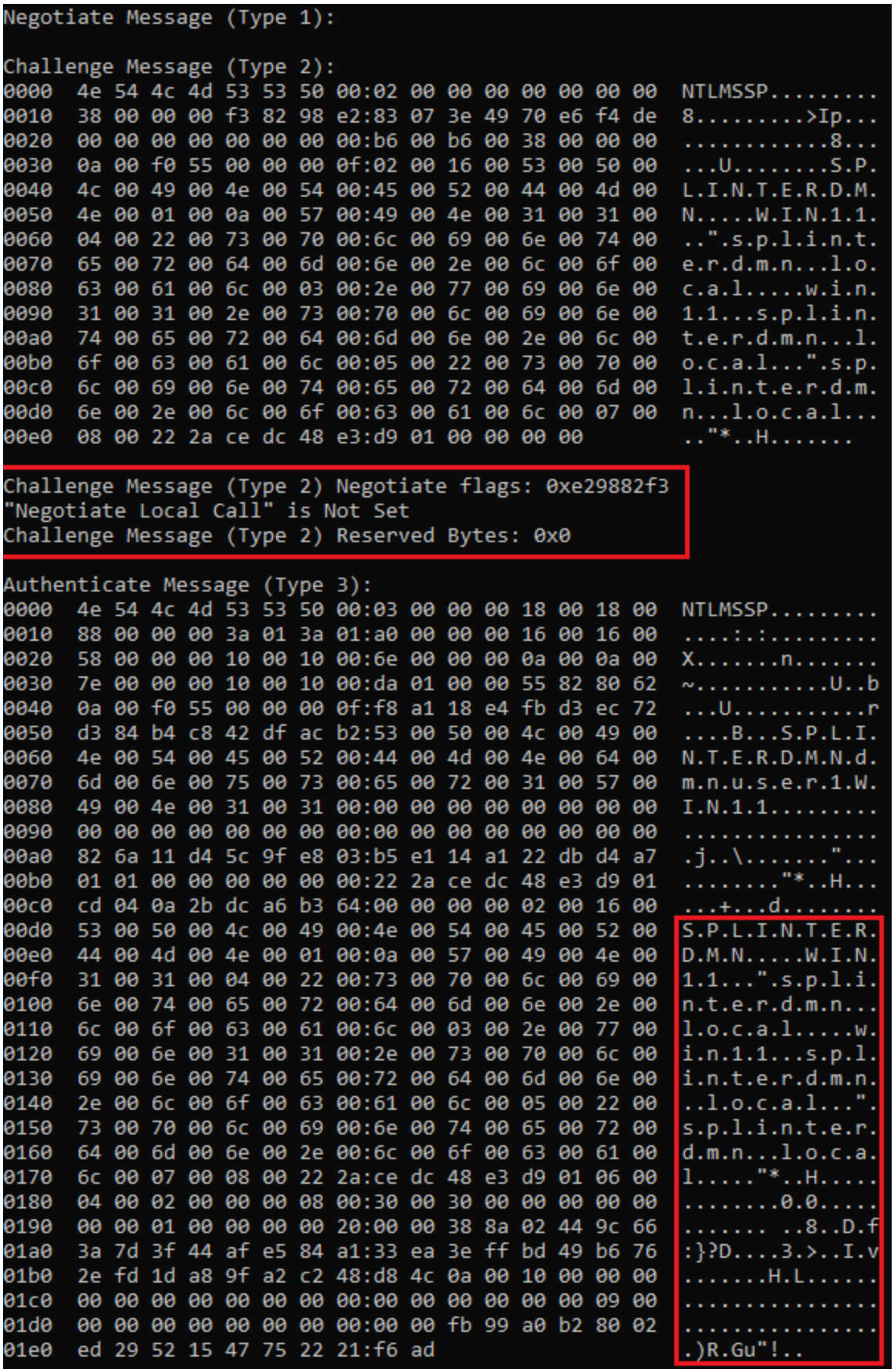
```
if ( (negotiateMessageFlags & NTLMSSP_NEGOTIATE_128) != 0 )
{
  challengeMessage->NegotiateFlags |= NTLMSSP_NEGOTIATE_128;
  negotiateMessageFlags = NegotiateMessage->NegotiateFlags;
}
// check if the client is running locally if it has provided both Domain Name and User Name and has not opted for datagram context
if ( (negotiateMessageFlags & NTLMSSP_NEGOTIATE_DATAGRAM) != 0
  || (negotiateMessageFlags & (NTLMSSP_NEGOTIATE_OEM_WORKSTATION_SUPPLIED|NTLMSSP_NEGOTIATE_OEM_DOMAIN_SUPPLIED)) != 0x3000 )
{
  goto SkipLocalAuth;
}
```

SsprHandleNegotiateMessage reversed code that checks Negotiate flags
So the function is checking the Negotiate flags supplied by the client and specifically checks if NTLMSSP_NEGOTIATE_OEM_WORKSTATION_SUPPLIED and NTLMSSP_NEGOTIATE_OEM_DOMAIN_SUPPLIED are set, which is always true if you use SSPI in the latest Windows versions.

However, what in the hell is the other checked flag NTLMSSP_NEGOTIATE_DATAGRAM ? Googling around brought me to Datagram Contexts.
I still haven't understood what is the intended behavior usage for this feature, but all i needed to know is that i can set this "mode" from the client by using the flag ISC_REQ_DATAGRAM in the first InitializeSecurityContext client call. Hopefully, by doing so, i would have forced the intended network auth i was aiming for.

The only thing to take into consideration is that mode is using **connection-less context semantics** and could be problematic to synchronize with external services. But… for our case we can run the server and client within the **same process** and we should be good. Even if it sounds very weird, it's what we need… In the end we just need to **trick** LSASS to **forge** the token for us.

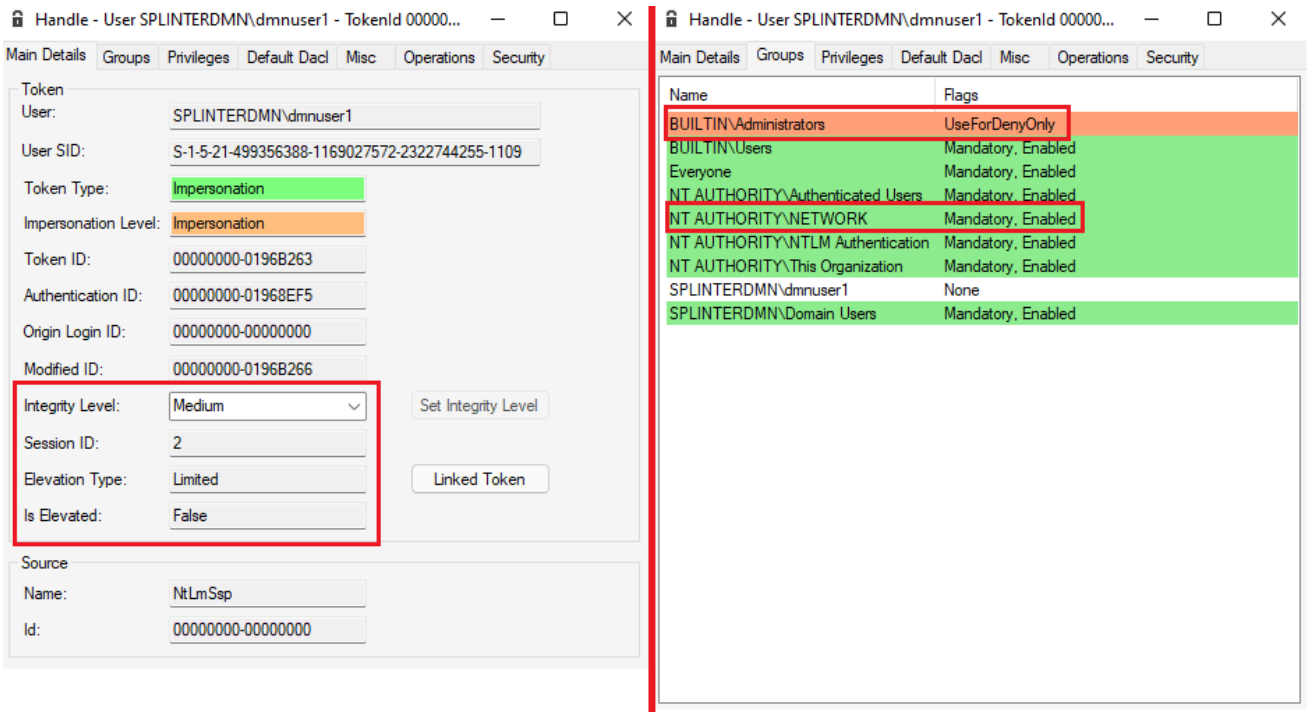
Let's sort out all of the code and check how the generated security buffers appears while using **Datagram Contexts**:



NTLM message exchanges with Datagram Contexts

Observing the security buffers exchanged, we can see that the “Negotiate Local Flag” is not set and that the “Reserved” bytes are 0, so no context handle has been passed by the server. Moreover, the client also sent the NTLMv2 Response in the Authenticate message. It definitely looks like the client and server are not negotiating a local authentication.
Note that the Negotiate message (Type 1) generated in Datagram-style authentication is empty and this is one of the significant differences compared to “normal” connection-oriented authentications.

Let’s inspect the token generated by this authentication and specifically if it contains the magic NETWORK SID logon type:



TokenViewer view of the generated token from Datagram-style authentication
The good news is that the NETWORK SID has been added in our token, so mission accomplished.

The very bad news is that somehow the token has been filtered by UAC. As you can see, the IL of the token is Medium and is not even Elevated.
My assumption that Local Authentication is the only mechanism to filter tokens is wrong. Probably, LSASS has additional checks in place, which i don’t plan to discover.

GAME OVER.

Sharing a logon session a little too much, part 2

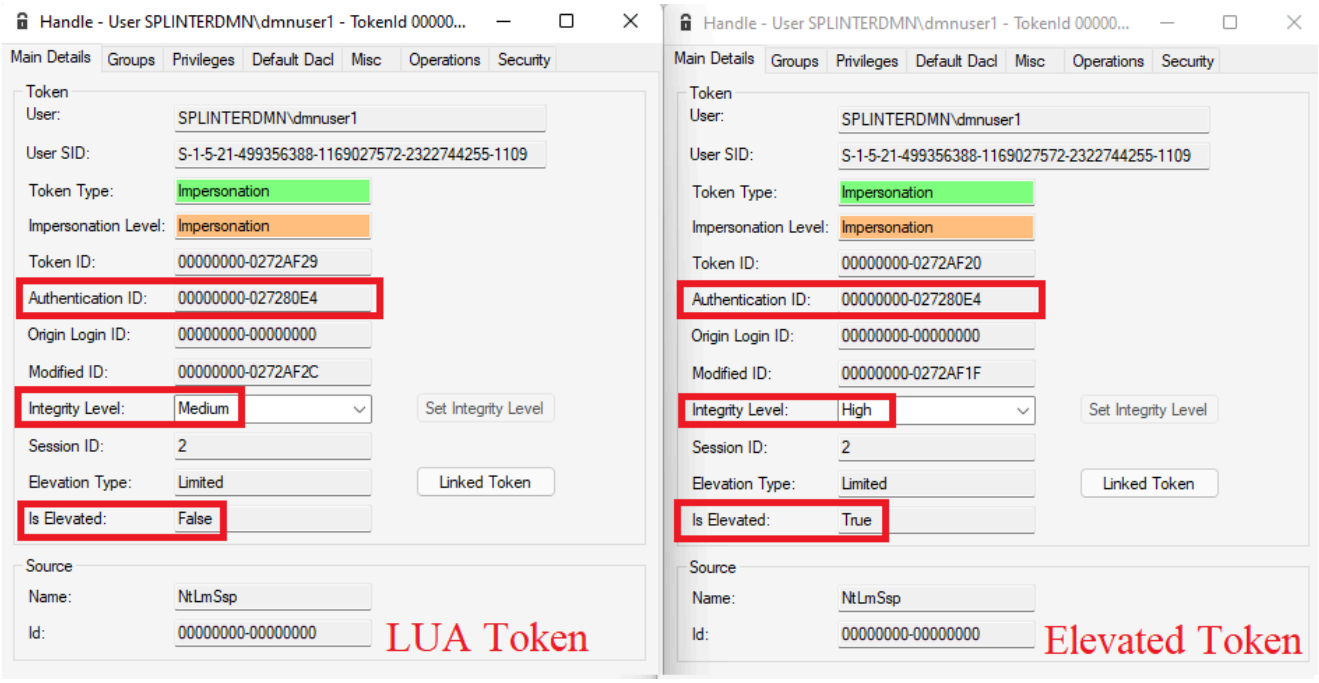
After almost 2 years from my last defeat against UAC, i decided to look again into this abandoned idea.

This time instead, i recalled the blog post “Sharing a Logon Session a Little Too Much” by James Forshaw and it inspired me for a **different exploitation path**.

What stands out from his blogpost is that when you do a **loopback network authentication** you can exploit a behavior of AcquireCredentialsHandle when used in network redirectors in which would result in LSASS using the first token created in the logon session rather than the caller’s token.

How that would apply in our case?

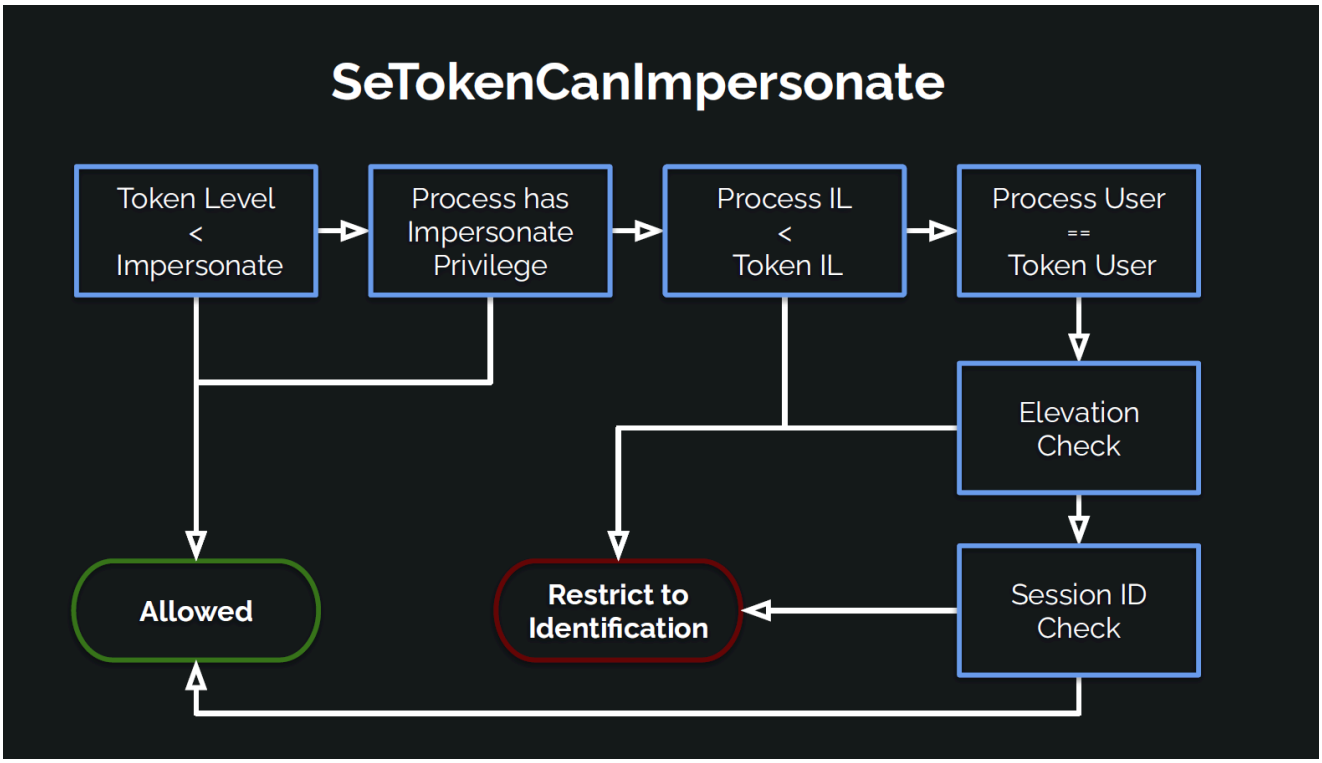
When we complete a **Datagram-style authentication**, LSASS creates a new logon session and creates the elevated token. Then, starting from the elevated token will create a new **filtered** token (LUA token) and the two are linked. The LUA token is the one actually associated with the security context “sent” to the server.



LUA Token vs. Elevated Token properties differences

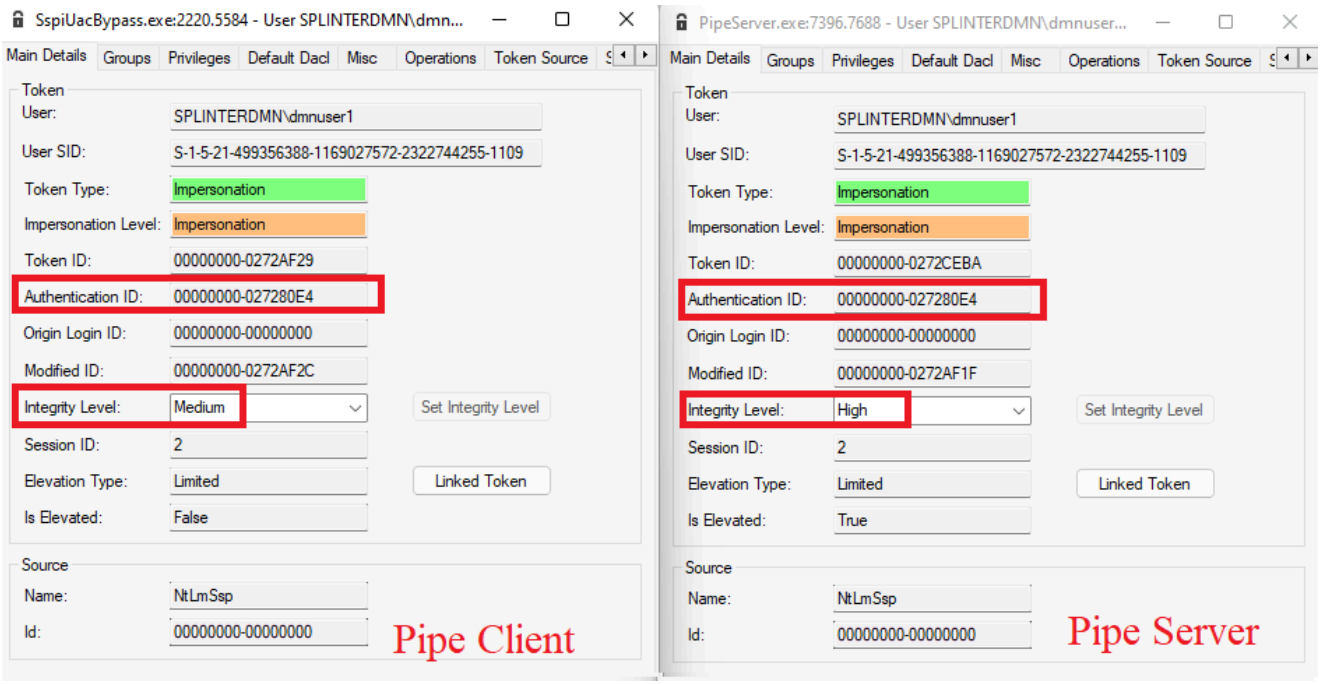
In the tokens generated in this way, the Logon Session ID (or Authentication ID from the token perspective) are equals and the Token ID values suggest that the Elevated Token is created before and likely is the first token created in that logon session. So according to this "token confusion" bug in LSASS, the server would see our call as it was **originating** from our **elevated** token rather than our impersonated limited token.

To exploit this bug, we first need to check if we are able to impersonate the generated LUA token.
According to Microsoft documentation of ImpersonateLoggedOnUser function we should be fine in impersonating a token as long as "the authenticated identity is same as the caller", which is our case. However, it's not entirely true. There are more conditions in place in the kernel function SeTokenCanImpersonate that is performing the checks:



SeTokenCanImpersonate flow for impersonation decisions, from "Taking Kerberos to The Next Level"
Comparing the token properties with our process's token running under UAC limitations, all conditions appear to be met.

Cool! So let's impersonate the token from the Datagram-style authentication and try to write to a named pipe over the **loopback interface**, e.g. \\127.0.0.1.\pipe\dummypipe



Pipe client thread vs. Pipe server thread tokens

Aaand BAM! We are able to **authenticate** over the loopback interface with our **elevated** token even if we are **impersonating** the **filtered token**!

Of course the pipe server is running with elevated privileges, otherwise the High IL token would have been downgraded to an Identification token.
But what about using this token for authenticating to an already running privileged service? Like the file-sharing service over SMB? It should be as easy as invoking CreateFile using an **UNC path**, like \\127.0.0.1\C$\Windows\bypassuac.txt

It worked!

So at this point we have a **privileged file write** primitive that can be combined with any known DLL Hijacking technique to achieve EoP, such as using an XPS Print Job or NetMan DLL Hijacking.

Privileged File Write is good but Code Execution is better :D

If you remember, i previously showed you that i'm able to authenticate even to a **named pipe** with the **elevated** token.
Having privileged access to named pipes means we have access to all of the RPC servers

running with ncacn_np configuration, which are a lot!
So, why we don't leverage this bug/feature to achieve **code execution** instead of our current privileged file write? We have a lot of juicy candidates like Remote SCM, Remote Registry, Remote Task Scheduler and so on…

However, if we try to authenticate to the **Remote Registry** through a RegConnectRegistryW call, it will **fail** to open handles to privileged regkeys.
Let's inspect the behavior:

```
Breakpoint 0 hit
SSPICLI!AcquireCredentialsHandleW:
0033:00007ffa`182bb070 4c8bdc          mov     r11,rsp
0: kd> k
 # Child-SP          RetAddr               Call Site
00 00000032`86d2f618 00007ffa`1b2d1095     SSPICLI!AcquireCredentialsHandleW
01 00000032`86d2f620 00007ffa`1b2c7574     RPCRT4!SECURITY_CREDENTIALS::AcquireCredentialsForClient+0x141
02 00000032`86d2f6b0 00007ffa`1b23bef8     RPCRT4!BINDING_HANDLE::SetAuthInformation+0x3e4
03 00000032`86d2f730 00007ffa`1b26fc71     RPCRT4!RpcBindingSetAuthInfoExW+0x238
04 00000032`86d2f830 00007ffa`1a6b4f17     RPCRT4!RpcBindingSetAuthInfoW+0x21
05 00000032`86d2f880 00007ffa`1a6b5216     ADVAPI32!BaseBindToMachine+0x1a3
06 00000032`86d2f8f0 00007ff7`dd451177     ADVAPI32!RegConnectRegistryExW+0xc6
07 00000032`86d2f980 00007ff7`dd4517e0     SspiUacBypass!main+0x107 [C:\Users\splintercode\source\repos\Ss
08 (Inline Function) --------`--------     SspiUacBypass!invoke_main+0x22 [D:\a\_work\1\s\src\vctools\crt\
09 00000032`86d2f9e0 00007ffa`1b0455a0     SspiUacBypass!__scrt_common_main_seh+0x10c [D:\a\_work\1\s\src\
0a 00000032`86d2fa20 00007ffa`1bb2485b     KERNEL32!BaseThreadInitThunk+0x10
0b 00000032`86d2fa50 00000000`00000000     ntdll!RtlUserThreadStart+0x2b
0: kd> dq r9
00000000`00000000  ????????`???????? ????????`????????
```

WinDbg details of AcquireCredentialsHandle call from RegConnectRegistryW

What it turns out is that the RPC runtime library (RPCRT4.dll) uses his **own implementation** for the authentication. As we can observe, the pvLogonId parameter for AcquireCredentialsHandleW is set to 0 which wouldn't allow to trigger the bug in LSASS and would use the **proper limited token** for the auth.
Now let's see the difference when authenticating to the loopback interface with the CreateFileW function:

WinDbg details of AcquireCredentialsHandle call from CreateFileW

The first difference we see here is that the authentication is implemented in the **kernel** by the SMB redirector driver **mrxsmb20.sys**.

More important, the pvLogonId parameter for AcquireCredentialsHandleW is set to the **logon session** associated with our user, which is what would fool lsass in **using** the **elevated token** from that logon session.
According to the documentation, in order to specify the pvLogonId you need to have the **SeTcbPrivilege**, which is not a problem in this case due to the fact that the code is running with **kernel privileges**.

This means, unfortunately, we can't use the **RPC runtime library** to authenticate to named pipes associated with RPC services if we want to exploit this bug.
However, no one could prohibit us to use our own **custom RPC client** implementation that leverages the **CreateFileW** call for authenticating to the RPC service over SMB. But this would require some hard work and i'm too lazy for that.

But this time luck seems to have been turned to my side and i found out @x86matthew already did this for the **service control manager** RPC interface in CreateSvcRpc!
The only change we need to do is to force the usage of **SMB** instead of **ALPC**, that technically translates in changing the pipe path from \\.\pipe\ntsvcs to \\127.0.0.1\pipe\ntsvcs

Let's see the full chain in action 😎

```
C:\uacbypass>whoami /groups | findstr Medium
Mandatory Label\Medium Mandatory Level                      Label            S-1-16-8192


C:\uacbypass>echo test > C:\Windows\bypassuac.txt
Access is denied.

C:\uacbypass>SspiUacBypass.exe

        SspiUacBypass - Bypassing UAC with SSPI Datagram Contexts
        by @splinter_code

Forging a token from a fake Network Authentication through Datagram Contexts
Network Authentication token forged correctly, handle --> 0x138
Forged Token Session ID set to 2. lsasrv!LsapApplyLoopbackSessionId adjusted the token to our current session
Bypass Success! Now impersonating the forged token... Loopback network auth should be seen as elevated now
Invoking CreateSvcRpc (by @x86matthew)
Connecting to \\127.0.0.1\pipe\ntsvcs RPC pipe
Opening service manager...
Creating temporary service...
Executing 'cmd /c "echo SspiUacBypass > C:\Windows\bypassuac.txt"' as SYSTEM user...
Deleting temporary service...
Finished

C:\uacbypass>type C:\Windows\bypassuac.txt
SspiUacBypass
```

PoC source code can be found at → https://github.com/antonioCoco/SspiUacBypass

Conclusion

A couple of years ago, i put this project between the many things i failed, thinking i hit a wall. Now i see the way was always there... I just needed to look at it differently or with a different perspective. It turned out to be a new cool way to get around UAC.

A big shout-out to James Forshaw and @x86matthew whose research provided invaluable insights and my friend @decoder_it for the proofread!

That's all folks, see you next time 👏

References

- ZeroNights 2017 James Forshaw Abusing Access Tokens for UAC Bypasses
- Reading Your Way Around UAC (Part 1)
- Reading Your Way Around UAC (Part 2)
- Reading Your Way Around UAC (Part 3)
- Farewell to the Token Stealing UAC Bypass
- Accessing Access Tokens for UIAccess
- Bypassing UAC in the most Complex Way Possible!
- Sharing a Logon Session a Little Too Much
- CreateSvcRpc - A custom RPC client to execute programs as the SYSTEM user