LAPS 2.0 Internals - XPN InfoSec Blog

« Back to home

LAPS 2.0 Internals

For most security consultant out there working with Windows environments, you've probably found yourself writing the same recommendation over the years:

"Shared local Administrator password… Deploy LAPS"

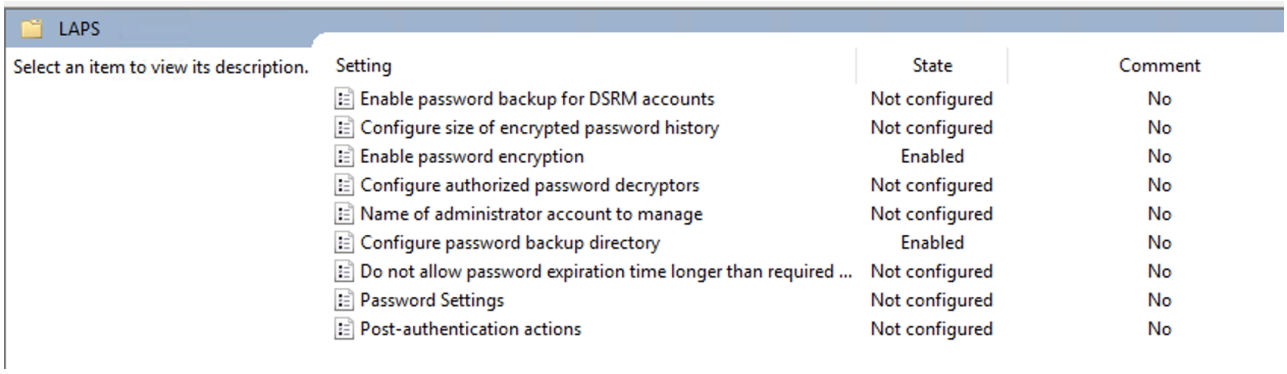
This year, LAPS 2.0 was released by Microsoft, and thankfully it now comes built-in to Windows. This time it comes ready for use with Active Directory, as well as being supported in Azure AD aka Entra ID. In this post, we'll expand on a Twitter thread that I posted on how LAPS 2.0 for Active Directory works under the hood, so you can make those fresh recommendations to your clients, and prepare yourself for the inevitable question: "But we just deployed LAPS.. what does LAPS 2.0 do differently?!".

Lab Setup

Before we can explore LAPS 2.0, we'll need a lab. Let's set up a quick environment to play around with.

To set up LAPS 2.0 in Active Directory, we need to make sure we are running the April 2023 security update. We can enable the support through Group Policy in `Computer Settings` -> `Administrative Templates` -> `System` -> `LAPS`.

We need to set `Configure password backup directory` to `Active Directory` to enable LAPS, and we'll enable `Enable Password Encryption` to take advantage of the new password encryption option:



Next, we need to update the AD schema using the PowerShell command `Update-LAPSADSchema`:

```
PS C:\Users\itadmin> Update-LapsADSchema

The 'ms-LAPS-Password' schema attribute needs to be added to the AD schema.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): Y

The 'ms-LAPS-PasswordExpirationTime' schema attribute needs to be added to the AD schema.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):

The 'ms-LAPS-EncryptedPassword' schema attribute needs to be added to the AD schema.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):

The 'ms-LAPS-EncryptedPasswordHistory' schema attribute needs to be added to the AD schema.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):

The 'ms-LAPS-EncryptedDSRMPassword' schema attribute needs to be added to the AD schema.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):

The 'ms-LAPS-EncryptedDSRMPasswordHistory' schema attribute needs to be added to the AD schema.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):

The 'ms-LAPS-Encrypted-Password-Attributes' extended right needs to be added to AD.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):

The mayContain attribute on the 'computer' class schema attribute needs to be updated in the AD schema.
Do you want to proceed?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):
```

To allow computers to manage their newly created LDAP records for LAPS, we create a new OU which we will call `LAPSManaged` and add in any computer objects which should be handled by LAPS.

To grant access to the OU container to computers when handling LAPS, we use:

```
Set-LapsADComputerSelfPermission -Identity OU=LAPSManaged,DC=lab,DC=local
```

```
PS C:\Users\itadmin> Set-LapsADComputerSelfPermission -Identity "OU=LAPSManaged,DC=lab,DC=local"

Name        DistinguishedName
----        -----------------
LAPSManaged OU=LAPSManaged,DC=lab,DC=local
```

With permission now granted to computer accounts within the OU, we use the `Reset-LAPSPassword` cmdlet to force a password to be added to LAPS 2.0 in AD:

```
PS C:\Users\itadmin> Reset-LapsPassword -Verbose
VERBOSE: BeginProcessing started
VERBOSE: Verifying that current process is running as a local administrator
VERBOSE: Success: current process is running as a local administrator
VERBOSE: Verifying that current process is elevated
VERBOSE: Success: current process is elevated
VERBOSE: BeginProcessing completed
VERBOSE:
VERBOSE: ProcessRecord started
VERBOSE: Calling ResetPassword RPC method
VERBOSE: ResetPassword RPC method returned hr:0x0
VERBOSE: ResetPassword RPC method succeeded
VERBOSE: ProcessRecord completed
VERBOSE:
VERBOSE: EndProcessing started
VERBOSE: EndProcessing completed
VERBOSE:
PS C:\Users\itadmin>
```

Then we can use `Get-LAPSADPassword` to retrieve the password:

```
PS C:\Users\itadmin> Get-LapsADPassword -Identity WinClient


ComputerName        : WINCLIENT
DistinguishedName   : CN=WINCLIENT,OU=LAPSManaged,DC=lab,DC=local
Account             : Administrator
Password            : System.Security.SecureString
PasswordUpdateTime  : 8/9/2023 9:51:32 PM
ExpirationTimestamp : 9/8/2023 9:51:32 PM
Source              : EncryptedPassword
DecryptionStatus    : Success
AuthorizedDecryptor : lab\Domain Admins
```

That should be enough to get us started, now to reversing!

Reversing LAPS 2.0

We know from above that `Get-LapsADPassword` has the ability to retrieve plaintext LAPS 2.0 passwords, so it stands to reason that we would start there.

And as we're dealing with PowerShell, we'll fire up dnSpy. The DLL we need is found in:

`C:\Windows\System32\WindowsPowerShell\v1.0\Modules\LAPS`

Specifically we'll need the `lapspsh.dll` assembly.

If we search for the `Get-LAPSADPassword` cmdlet, we'll find a class called `Microsoft.Windows.LAPS.GetLapsADPassword`:

```
namespace Microsoft.Windows.LAPS
{
    // Token: 0x02000010 RID: 16
    [Cmdlet("Get", "LapsADPassword", DefaultParameterSetName = "NormalMode")]
    [CmdletRequiresDomainJoin]
    public sealed class GetLapsADPassword : CmdletBase, IDisposable
    {
```

In this class, we have the method `ProcessOneIdentity` which contains the logic for retrieving the LAPS 2.0 credentials from LDAP:

```
private void ProcessOneIdentity(string Identity)
{
    List<PasswordInfo> list = new List<PasswordInfo>();
    base.Verbose("Processing Identity:'{0}'", new object[] { Identity });
    ComputerNameInfo computerNameInfo = base.GetComputerNameInfo(this._ldapConn, this._ldapConnectionInfo, Identity);
    base.Verbose("Found computer object in AD: {0}", new object[] { computerNameInfo.ToString() });
    AccountPasswordAttributes passwordAttributes = base.GetPasswordAttributes(this._ldapConn, computerNameInfo.DistinguishedName);
    if (passwordAttributes == null)
    {
```

The call to `GetPasswordAttributes` shows the LDAP queries being used to query AD for passwords:

```
// Token: 0x0600002A RID: 42 RVA: 0x00003E90 File Offset: 0x00002090
protected AccountPasswordAttributes GetPasswordAttributes(LdapConnection ldapConn, string computerDN)
{
    string[] array = new string[] { "msLAPS-PasswordExpirationTime", "msLAPS-Password", "msLAPS-EncryptedPassword", "msLAPS-EncryptedPasswordHistory", "msLAPS-
      EncryptedDSRMPassword", "msLAPS-EncryptedDSRMPasswordHistory", "ms-Mcs-AdmPwd", "ms-Mcs-AdmPwdExpirationTime" };
    DateTime? dateTime = null;
    string text = null;
    byte[] array2 = null;
    byte[][] array3 = null;
    byte[] array4 = null;
    byte[][] array5 = null;
    string text2 = null;
    DateTime? dateTime2 = null;
    string text3 = CmdletBase.LdapEscape(computerDN);
    string text4 = string.Format(CultureInfo.InvariantCulture, "(&(objectClass={0})({1}={2}))", "computer", "distinguishedName", text3);
    this.Verbose("Querying computer object for password attributes:");
    this.Verbose("  escapedComputerDN: {0}", new object[] { text3 });
    this.Verbose("  ldapFilter:        {0}", new object[] { text4 });
    SearchRequest searchRequest = new SearchRequest(text3, text4, System.DirectoryServices.Protocols.SearchScope.Base, array);
    SearchResponse searchResponse = ldapConn.SendRequest(searchRequest) as SearchResponse;
    if (searchResponse.Entries.Count != 1)
    {
        return null;
    }
    SearchResultEntry searchResultEntry = searchResponse.Entries[0];
    this.Verbose("Search succeeded with {0} returned attributes:", new object[] { searchResultEntry.Attributes.Count });
```

From this method we find several new LDAP attributes being retrieved, but the ones that stand out for us are:

- `msLAPS-EncryptedPassword`
- `msLAPS-Password`

One of the new features of LAPS 2.0 is the support for encrypted credentials. Previously credentials for LAPS were stored in the plain-text attribute `ms-Mcs-AdmPwd`, but now it seems that that `msLAPS-EncryptedPassword` hosts the encrypted credentials, leaving `msLAPS-Password` to host the plain-text password if encryption hasn't been enabled in Group Policy.

To decrypt the stored passwords, there is a call being made to `BuildPasswordInfoFromEncryptedPassword` which again signals from the debug output that we are in the correct place:

```
// Token: 0x06000057 RID: 87 RVA: 0x00005804 File Offset: 0x00003A04
private PasswordInfo BuildPasswordInfoFromEncryptedPassword(ComputerNameInfo computerNameInfo, PasswordSource passwordSource, byte[] encryptedPassword,
  DateTime? passwordExpirationTimestampUTC, bool useRecoveryMode, bool useLocalKDS)
{
    DecryptionStatus decryptionStatus;
    EncryptedPasswordAttributeState encryptedPasswordAttributeState = base.ParseAndDecryptDirectoryPassword(this._hDecryptionToken, encryptedPassword,
      useRecoveryMode, useLocalKDS, out decryptionStatus);
    string text;
    string text2;
    DateTime? dateTime;
    if (decryptionStatus == DecryptionStatus.Success)
    {
        base.Verbose("Successfully decrypted inner password buffer");
        text = encryptedPasswordAttributeState.InnerState.AccountName;
```

The method `ParseAndDecryptDirectoryPassword` is passed the encrypted data retrieved from the `msLAPS-EncryptedPassword` attribute:

```
// Token: 0x06000031 RID: 49 RVA: 0x0000466C File Offset: 0x0000286C
protected EncryptedPasswordAttributeState ParseAndDecryptDirectoryPassword(IntPtr hDecryptionToken, byte[] encryptedPasswordBytes, bool useRecoveryMode, bool useLocalKDS, out
  DecryptionStatus decryptionStatus)
{
    byte[] array = null;
    EncryptedPasswordAttributePrefixInfo encryptedPasswordAttributePrefixInfo = EncryptedPasswordAttributePrefixInfo.ParseFromBuffer(encryptedPasswordBytes);
    byte[] array2 = new byte[encryptedPasswordAttributePrefixInfo.EncryptedBufferSize];
    Buffer.BlockCopy(encryptedPasswordBytes, 16, array2, 0, (int)encryptedPasswordAttributePrefixInfo.EncryptedBufferSize);
    string text = this.ExtractAndResolveSidProtectionString(array2);
    byte[] array3;
    uint num = this.DecryptBytesHelper(hDecryptionToken, array2, useRecoveryMode, useLocalKDS, out array3);
    EncryptedPasswordAttributeInner encryptedPasswordAttributeInner;
    if (num == 0U)
    {
        encryptedPasswordAttributeInner = EncryptedPasswordAttributeInner.ParseFromJson(Encoding.Unicode.GetString(array3));
        decryptionStatus = DecryptionStatus.Success;
    }
    else if (num == 2148073516U)
    {
        encryptedPasswordAttributeInner = null;
        decryptionStatus = DecryptionStatus.Unauthorized;
    }
    else
    {
        encryptedPasswordAttributeInner = null;
        decryptionStatus = DecryptionStatus.Error;
    }
    uint num2 = (uint)(encryptedPasswordBytes.Length - 16 - (int)encryptedPasswordAttributePrefixInfo.EncryptedBufferSize);
    if (num2 > 0U)
    {
        array = new byte[num2];
        Buffer.BlockCopy(encryptedPasswordBytes, (int)(16U + encryptedPasswordAttributePrefixInfo.EncryptedBufferSize), array, 0, (int)num2);
    }
    return new EncryptedPasswordAttributeState(text, encryptedPasswordAttributePrefixInfo, encryptedPasswordAttributeInner, array);
}
```

Reviewing this method, we find that the first 16 bytes are used as a “prefix”. The fields in this prefix are:

- Bytes 0 - 3 = Upper Date Time Stamp
- Bytes 4 - 7 = Lower Date Time Stamp
- Bytes 8 - 11 = Encrypted Buffer Size
- Bytes 12 - 15 = Flags

The method then skips over this 16 byte prefix, and decrypts the remaining contents using the method `DecryptBytesHelper`.

`DecryptBytesHelper` is just a wrapper to invoke the function `DecryptNormalMode`, which is present in another DLL:

```
// Token: 0x06000070 RID: 112
[DllImport("lapsutil.dll", CharSet = CharSet.Unicode)]
public static extern uint DecryptNormalMode(IntPtr hDecryptionIdentityToken, IntPtr pbData, uint cbData, uint ulFlags, out IntPtr pbDecryptedData, out uint cbDecryptedData);

// Token: 0x06000071 RID: 113
[DllImport("lapsutil.dll", CharSet = CharSet.Unicode)]
public static extern uint DecryptRecoveryMode(IntPtr pbData, uint cbData, out IntPtr pbDecryptedData, out uint cbDecryptedData);
```

To reverse this DLL, we need to transition over to Ghidra for disassembly, as these methods are defined in native DLL’s rather than .NET.

Looking at the exported `DecryptNormalMode` function, we see that this is a wrapper around calls to `NCrypt` where the encrypted blob passed from .NET is decrypted:

```
local_58 = BufferEncryptDecryptCallback;
local_50 = &local_48;
local_40 = ZEXT816(0);
iVar3 = NCryptStreamOpenToUnprotect(&local_58,flags | 0x40,0,&local_res10
if (iVar3 == 0) {
  if ((((*(uint *)(WPP_GLOBAL_Control + 0x1c) & 0x200) != 0) &&
      (4 < (byte)WPP_GLOBAL_Control[0x19])) {
    WPP_SF_(*(undefined8 *)(WPP_GLOBAL_Control + 0x10),0x18,
            &WPP_3545a8ee8f10384d6a180lee0a424267_Traceguids);
  }
  iVar3 = NCryptStreamUpdate(local_res10,pbData,cbData,1);
  if (iVar3 == 0) {
    if ((((*(uint *)(WPP_GLOBAL_Control + 0x1c) & 0x200) != 0) &&
        (4 < (byte)WPP_GLOBAL_Control[0x19])) {
      WPP_SF_(*(undefined8 *)(WPP_GLOBAL_Control + 0x10),0x1b,
              &WPP_3545a8ee8f10384d6a180lee0a424267_Traceguids);
    }
    puVar2 = (uchar *)local_40._0_8_;
    auVar1._8_8_ = 0;
    auVar1._0_8_ = local_40._8_8_;
    local_40 = auVar1 << 0x40;
    *cbDecryptedData = puVar2;
    *pbDecryptedData = (int)local_30 - (int)local_28;
    lVar4 = 0;
  }
```

The decrypted content is then returned to .NET where it is parsed as JSON and the password is recovered:

```
// Token: 0x0200003A RID: 58
public class EncryptedPasswordAttributeInner
{
    // Token: 0x060000FB RID: 251 RVA: 0x00008A4C File Offset: 0x00006C4C
    public static EncryptedPasswordAttributeInner ParseFromJson(string passwordJson)
    {
        EncryptedPasswordAttributeRaw encryptedPasswordAttributeRaw = EncryptedPasswordAttributeRaw.Parse(passwordJson);
        if (string.IsNullOrEmpty(encryptedPasswordAttributeRaw.AccountName))
        {
            throw new ArgumentException("AccountName field was missing from encrypted attribute");
        }
        if (string.IsNullOrEmpty(encryptedPasswordAttributeRaw.UpdateTimestamp))
        {
            throw new ArgumentException("UpdateTimestamp field was missing from encrypted attribute");
        }
        if (string.IsNullOrEmpty(encryptedPasswordAttributeRaw.Password))
        {
            throw new ArgumentException("Password field was missing from encrypted attribute");
        }
        DateTime dateTime = DateTime.FromFileTimeUtc(long.Parse(encryptedPasswordAttributeRaw.UpdateTimestamp, NumberStyles.HexNumber));
        return new EncryptedPasswordAttributeInner(encryptedPasswordAttributeRaw.AccountName, encryptedPasswordAttributeRaw.Password, dateTime);
    }
```

Recovery with .NET

Hopefully with this quick walkthrough you can see how we can recreate the process using a standalone .NET tool which will pull down and decrypt LAPS 2.0 credentials.

The first thing that we do is retrieve the LDAP entry:

```
string filter = string.Format("(&(objectClass={0})({1}={2}))", "computer",
"distinguishedName", dn);

// Create a new ldap connection
LdapConnection ldapConnection = new LdapConnection(dc);
ldapConnection.SessionOptions.ProtocolVersion = 3;
ldapConnection.Bind();

SearchRequest searchRequest = new SearchRequest(dn, filter, SearchScope.Base,
attributeList);

SearchResponse searchResponse = ldapConnection.SendRequest(searchRequest) as
SearchResponse;

SearchResultEntry searchResultEntry = searchResponse.Entries[0];
if (searchResponse.Entries.Count != 1)
{
    Console.WriteLine("[!] Could not find computer object");
    return;
}
```

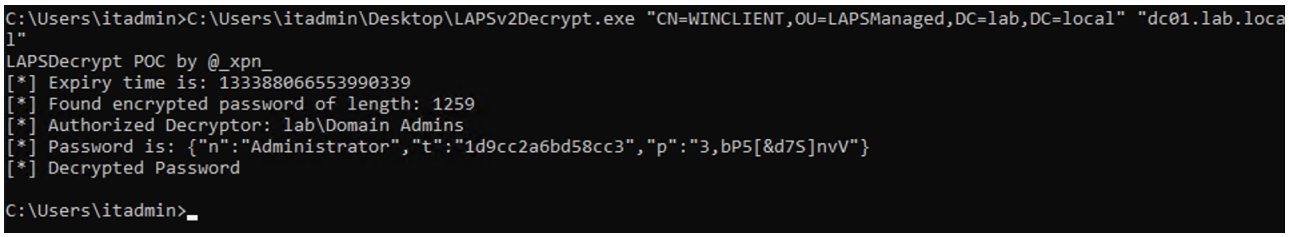
Once we've pulled down the LDAP record, we need to actually decrypt the LAPS 2.0 attribute value. We know that Microsoft are using the NCrypt calls to decrypt the content via `lapsutil.dll`, but we'll just use P/Invoke from .NET rather than calling out to an unmanaged DLL (there is also nothing stopping you from just using the `lapsutil.dll` if you want):

```
[DllImport("ncrypt.dll")]
public static extern uint NCryptStreamOpenToUnprotect(in NCRYPT_PROTECT_STREAM_INFO
pStreamInfo, ProtectFlags dwFlags, IntPtr hWnd, out IntPtr phStream);

[DllImport("ncrypt.dll")]
public static extern uint NCryptStreamUpdate(IntPtr hStream, IntPtr pbData, int
cbData, [MarshalAs(UnmanagedType.Bool)] bool fFinal);

[DllImport("ncrypt.dll")]
public static extern uint NCryptUnprotectSecret(out IntPtr phDescriptor, Int32
dwFlags, IntPtr pbProtectedBlob, uint cbProtectedBlob, IntPtr pMemPara, IntPtr hWnd,
out IntPtr ppbData, out uint pcbData);
```

By calling the NCrypt API from .NET, we can just decrypt the contents directly:

```
uint ret = Win32.NCryptStreamOpenToUnprotect(info, ProtectFlags.NCRYPT_SILENT_FLAG,
IntPtr.Zero, out handle);
if (ret == 0)
{
    IntPtr alloc = Marshal.AllocHGlobal(encryptedPass.Length);
    Marshal.Copy(encryptedPass, 16, alloc, encryptedPass.Length - 16);

    // Get the authorized decryptor of the blob
    ret = Win32.NCryptUnprotectSecret(out handle2, 0x41, alloc,
(uint)encryptedPass.Length - 16, IntPtr.Zero, IntPtr.Zero, out secData, out
secDataLen);
    if (ret == 0)
    {
    string sid;

        ret = NCryptGetProtectionDescriptorInfo(handle2, IntPtr.Zero, 1, out sid);
        if (ret == 0)
        {
            SecurityIdentifier securityIdentifier = new
SecurityIdentifier(sid.Substring(4, sid.Length - 4));

            try
            {
                ntaccount = (securityIdentifier.Translate(typeof(NTAccount)) as
NTAccount);

                Console.WriteLine("[*] Authorized Decryptor: {0}",
ntaccount.ToString());
            } catch
            {
                Console.WriteLine("[*] Authorized Decryptor SID: {0}",
securityIdentifier.ToString());
            }
        }
    }

    // Decrypt the blob
    ret = Win32.NCryptStreamUpdate(handle, alloc, encryptedPass.Length - 16, true);
    Console.WriteLine("[*] Decrypted Password");
}
```

And when run, we see the result:



The full code for this can be found here.

Once More with Feeling

For you users of `execute-assembly`, the above code should be fine. But if you prefer a BOF, we'll need to do this again in C++. Thankfully as I'm writing this post, Cobalt Strike just released the new BOF template for Visual Studio:

> Catching up on some blogging this week... time to test out the new BOF template from @HenriNurmi pic.twitter.com/gBAxr7c1v3
>
> — Adam Chester (@_xpn_) August 11, 2023

It's easy to port things over to native code now that we've mapped out the process:

```
beacon> lapsv2decrypt dc01.lab.local "dc=lab,dc=local" "CN=WINCLIENT,OU=LAPSManaged,DC=lab,DC=local"
[*] Target Server: dc01.lab.local
[*] Target DN: dc=lab,dc=local
[*] Computer DN: CN=WINCLIENT,OU=LAPSManaged,DC=lab,DC=local
[+] host called home, sent: 3581 bytes
[+] received output:
LAPSv2 Blob Header Info:
Upper Date Timestamp: 31050794
Lower Date Timestamp: 1809157315
Encrypted Buffer Size: 1243
Flags: 0
[+] received output:
Decrypted Output: {"n":"Administrator","t":"1d9cc2a6bd58cc3","p":"3,bP5[&d7S]nvV"}
```

I've added the C++ BOF code here. Have phun!