

Dumping Stored Credentials with SeTrustedCredmanAccessPrivilege

 tiraniddo.dev/2021/05/dumping-stored-credentials-with.html

I've been going through the various token privileges on Windows trying to find where they're used. One which looked interesting is SeTrustedCredmanAccessPrivilege which is documented as "*Access Credential Manager as a trusted caller*". The Credential Manager allows a user to store credentials, such as web or domain accounts in a central location that only they can access. It's protected using DPAPI so in theory it's only accessible when the user has authenticated to the system. The question is, what does having *SeTrustedCredmanAccessPrivilege* grant? I couldn't immediately find anyone who'd bothered to document it, so I guess I'll have to do it myself.

The Credential Manager is one of those features that probably sounded great in the design stage, but does introduce security risks, especially if it's used to store privileged domain credentials, such as for remote desktop access. An application, such as the remote desktop client, can store domain credential using the CredWrite API and specifying the username and password in the CREDENTIAL structure. The type of credentials should be set to `CRED_TYPE_DOMAIN_PASSWORD`.

An application can then access the stored credentials for the current user using APIs such as CredRead or CredEnumerate. However, if the type of credential is `CRED_TYPE_DOMAIN_PASSWORD` the *CredentialBlob* field which should contain the password is always empty. This is an artificial restriction put in place by LSASS which implements the credential manager RPC service. If a domain credentials type is being read then it will never return the password.

How does the domain credentials get used if you can't read the password? Security packages such as NTLM/Kerberos/TSSSP which are running within the LSASS process can use an internal API which doesn't restrict the reading of the domain password. Therefore, when you authenticate to the remote desktop service the target name is used to lookup available credentials, if they exist the user will be automatically authenticated.

The credentials are stored in files in the user's profile encrypted with the user's DPAPI key. Why can we not just decrypt the file directly to get the password? When writing the file LSASS sets a system flag in the encrypted blob which makes the DPAPI refuse to decrypt the blob even though it's still under a user's key. Only code running in LSASS can call the DPAPI to decrypt the blob.

If we have administrator privileges getting access the password is trivial. Read the [Mimikatz wiki page](#) to understand the various ways that you can use the tool to get access to the credentials. However, it boils down to one of the following approaches:

1. Patch out the checks in LSASS to not blank the password when read from a normal user.
2. Inject code into LSASS to decrypt the file or read the credentials.
3. Just read them from LSASS's memory.
4. Reimplement DPAPI with knowledge of the user's password to ignore the system flag.
5. Play games with the domain key backup protocol.

For example, Nirsoft's [CredentialsFileView](#) seems to use the injection into LSASS technique to decrypt the DPAPI protected credential files. (~~Caveat, I've only looked at v1.07 as v1.10 seems to not be available for download anymore, so maybe it's now different.~~ UPDATE: it seems available for download again but Defender thinks it's malware, plus ça change).

At this point you can probably guess that *SeTrustedCredmanAccessPrivilege* allows a caller to get access to a user's credentials. But how exactly? Looking at *LSASRV.DLL* which contains the implementation of the Credential Manager the privilege is checked in the function *CredplsRpcClientTrusted*. This is only called by two APIs, *CredrReadByTokenHandle* and *CredrBackupCredentials* which are exported through the *CredReadByTokenHandle* and *CredBackupCredentials* APIs.

The *CredReadByTokenHandle* API isn't that interesting, it's basically *CredRead* but allows the user to read from to be specified by providing the user's token. As far as I can tell reading a domain credential still returns a blank password. *CredBackupCredentials* on the other hand is interesting. It's the API used by *CREDWIZ.EXE* to backup a user's credentials, which can then be restored at a later time. This backup includes all credentials including domain credentials. The prototype for the API is as follows:

```
BOOL WINAPI CredBackupCredentials(HANDLE Token,  
                                  LPCWSTR Path,  
                                  PVOID Password,  
                                  DWORD PasswordSize,  
                                  DWORD Flags);
```

The backup process is slightly convoluted, first you run *CREDWIZ* on your desktop and select backup and specify the file you want to write the backup to. When you continue with the backup the process makes an RPC call to your *WinLogon* process with the credentials path which spawns a new copy of *CREDWIZ* on the secure desktop. At this point you're instructed to use CTRL+ALT+DEL to switch to the secure desktop. Here you type the password, which is used to encrypt the file to protect it at rest, and is needed when the

credentials are restored. *CREDWIZ* will even ensure it meets your system's password policy for complexity, how generous.

CREDWIZ first stores the file to a temporary file, as LSASS encrypts the encrypted contents with the system DPAPI key. The file can be decrypted then written to the final destination, with appropriate impersonation etc.

The only requirement for calling this API is having the *SeTrustedCredmanAccessPrivilege* privilege enabled. Assuming we're an administrator getting this privilege is easy as we can just borrow a token from another process. For example, checking for what processes have the privilege shows obviously *WinLogon* but also LSASS itself even though it arguably doesn't need it.

```
PS> $ts = Get-AccessibleToken
```

```
PS> $ts | ? {
```

```
    "SeTrustedCredmanAccessPrivilege" -in $_.ProcessTokenInfo.Privileges.Name
}
```

```
TokenId Access
```

```
Name
```

```
-----
```

```
1A41253 GenericExecute|GenericRead Lsalso.exe:124
```

```
1A41253 GenericExecute|GenericRead lsass.exe:672
```

```
1A41253 GenericExecute|GenericRead winlogon.exe:1052
```

```
1A41253 GenericExecute|GenericRead atieclxx.exe:4364
```

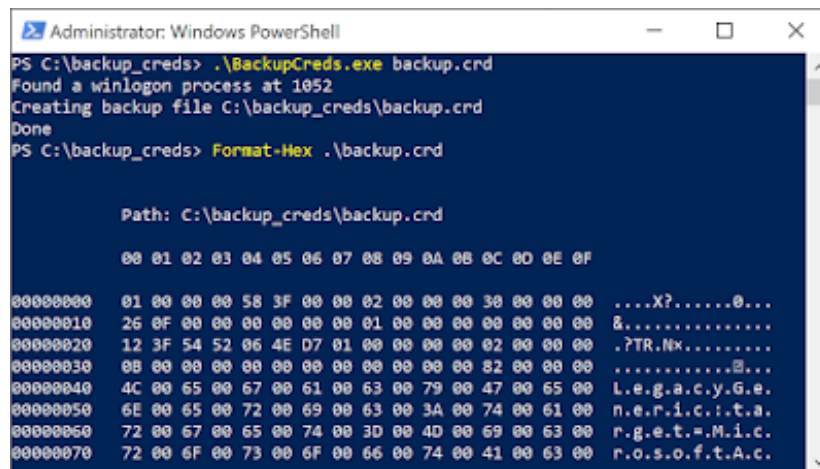
I've literally no idea what the *ATIECLXX.EXE* process is doing with *SeTrustedCredmanAccessPrivilege*, it's probably best not to ask ;-)

To use this API to backup a user's credentials as an administrator you do the following.

1. Open a *WinLogon* process for *PROCESS_QUERY_LIMITED_INFORMATION* access and get a handle to its token with *TOKEN_DUPLICATE* access.
2. Duplicate token into an impersonation token, then enable *SeTrustedCredmanAccessPrivilege*.
3. Open a token to the target user, who must already be authenticated.
4. Call *CredBackupCredentials* while impersonating the *WinLogon* token passing a path to write to and a NULL password to disable the user encryption (just to make life easier). It's *CREDWIZ* which enforces the password policy not the API.
5. While still impersonating open the file and decrypt it using the *CryptUnprotectData* API, write back out the decrypted data.

If it all goes well you'll have all the of the user's credentials in a packed binary format. I couldn't immediately find anyone documenting it, but people obviously have done before. I'll

leave doing all this yourself as a exercise for the reader. I don't feel like providing an implementation.



```
Administrator: Windows PowerShell
PS C:\backup_creds> .\BackupCreds.exe backup.crd
Found a winlogon process at 1052
Creating backup file C:\backup_creds\backup.crd
Done
PS C:\backup_creds> Format-Hex .\backup.crd

Path: C:\backup_creds\backup.crd

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 01 00 00 00 58 3F 00 00 02 00 00 00 30 00 00 00 ...X?.....0...
00000010 26 0F 00 00 00 00 00 00 01 00 00 00 00 00 00 00 &.....
00000020 12 3F 54 52 06 4E D7 01 00 00 00 00 02 00 00 00 .?TR.Nx.....
00000030 0B 00 00 00 00 00 00 00 00 00 00 00 82 00 00 00 .....@...
00000040 4C 00 65 00 67 00 61 00 63 00 79 00 47 00 65 00 L.e.g.a.c.y.G.e.
00000050 6E 00 65 00 72 00 69 00 63 00 3A 00 74 00 61 00 n.e.r.i.c.:.t.a.
00000060 72 00 67 00 65 00 74 00 3D 00 4D 00 69 00 63 00 r.g.e.t.=.M.i.c.
00000070 72 00 6F 00 73 00 6F 00 66 00 74 00 41 00 63 00 r.o.s.o.f.t.A.c.
```

Why would you do this when there already exists plenty of other options? The main advantage, if you can call it that, is you never touch LSASS and definitely never inject any code into it. This wouldn't be possible anyway if LSASS is running as PPL. You also don't need to access the SECURITY hive to extract DPAPI credentials or know the user's password (assuming they're authenticated of course). About the only slightly suspicious thing is opening *WinLogon* to get a token, though there might be alternative approaches to get a suitable token.