Le contenu de cette page n'est pas disponible dans la langue sélectionnée. Chez Elastic, nous mettons tout en œuvre pour vous proposer du contenu dans différentes langues. En attendant, nous vous remercions de votre patience ! ✕

20 AVRIL 2021   **PRODUCT**

How attackers abuse Access Token Manipulation (ATT&CK T1134)

Par **Will Burgess**

Partager

In our previous blog post on **Windows access tokens for security practitioners**, we covered:

- The relationship between logon sessions and access tokens
- How network authentication works in Windows environments

Having covered some of the key concepts in Windows security, we will now build on this knowledge and start to look at how attackers can abuse legitimate Windows functionality to move laterally and compromise Active Directory domains.

This blog has deliberately attempted to abstract away the workings of specific Windows network authentication protocols (e.g., NTLM and Kerberos) where possible. As a consequence, there may be instances where behaviour unique to these protocols differs with the behavior described below. It also assumes some basic understanding of the Kerberos authentication protocol[1].

Additionally, the material covered in this blog series was used for a BlackHat 2020 presentation, "Detecting Access Token Manipulation". The presentation can be found **here** and the slides **here**.

Access Token Manipulation (ATT&CK technique: T1134)

Having explained the basic principles of how logon sessions and access tokens work in **our previous blog post**, both locally and for distributed applications, this section will explain how attackers can abuse access tokens and target the fundamental trust relationships in Windows domains to compromise entire networks. The aim of this section is to describe access token manipulation **techniques** used by attackers within the *context* of a simulated compromise.

As a note, there is already an extensive body of excellent research on access token manipulation (which will be linked to liberally throughout this post). This blog attempts to build on this body of knowledge via considering access token manipulation from a different approach, namely through the relationship between access tokens, logon sessions and cached credentials. In the author's opinion, any description of token manipulation without considering these relationships represents only the tip

result is the same: the attacker's code is running in a process which has an access token belonging to the compromised user.

This means that any **local access checks** will use the compromised user's access token and any **remote authentication attempts** will use the compromised user's cached credentials[2]. Hence, the attacker can, both locally and across the network, perform *all* the actions that the compromised user can. For example, if any internal web applications use Windows SSO, an attacker will be able to access them as if they were the user.

Token Manipulation: The 'Art of the possible'

Typically, an attacker will want to move from the compromised endpoint to another host *as quickly as possible*[3]. When considering **lateral movement** from a token manipulation perspective, the attacker effectively has three options[4], each of which is constrained by the fundamental relationship between access tokens, logon sessions, and cached credentials, as illustrated below:
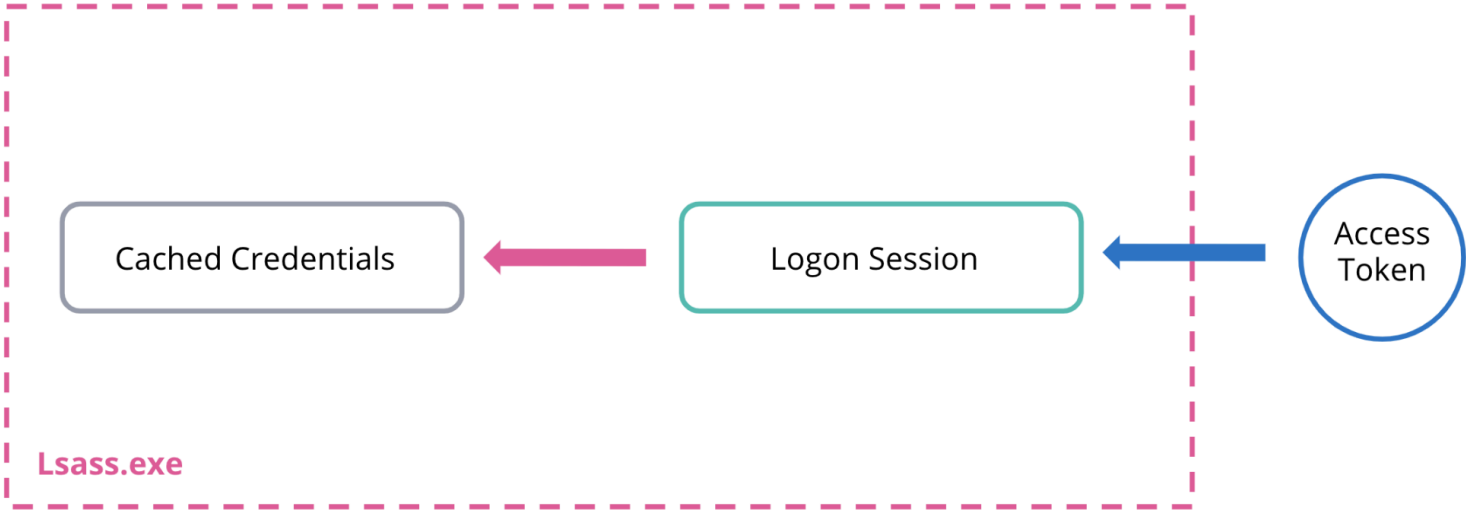

Figure 1 - The relationship between access tokens, logon sessions and cached credentials

If an attacker wants to move laterally via Windows SSO then all of these three links must be in place (e.g., they have a handle to a token which is linked to a logon session backed by their target credentials). Otherwise, an attacker's freedom of movement relies on either **creating** new links (e.g., new logon sessions) or **modifying** existing ones (e.g., changing cached credentials or the logon session that their access token points to). These constraints are discussed in more detail in the three options below:

1. Steal the token of an already logged-on privileged user (non-network logon)

If another privileged user is already logged on to the compromised host, an attacker can escalate their privileges and obtain a handle to an access token representing this user. Irrespective of whether the attacker impersonates the stolen token or starts a new process, if that token is linked to a *non-network* logon session, it will have cached credentials, and hence the attacker can auth off the box to another host[5]. Hence, this technique allows an attacker to use another user's credentials to access remote

finding-alternatives-to-winlogon-exe-000000007ob

- https://foxglovesecurity.com/2016/09/26/rotten-potato-privilege-escalation-from-service-accounts-to-system/

- https://labs.f-secure.com/assets/BlogFiles/mwri-security-implications-of-windows-access-tokens-2008-04-14.pdf

2. Create a new logon session with stolen credentials and impersonate the returned token or spawn a new process with it

In this case, there is no privileged user already logged on (and hence no corresponding **useful** access token/logon session), but the attacker still needs to find a way to *change their security context*.

Hence, the attacker must find credentials elsewhere and use these stolen credentials to create a **new** logon session as the compromised user. **As Windows will automatically cache credentials for certain logon types**, the attacker can now obtain a newly minted access token which is backed up by the stolen credentials. Once the attacker has a handle to a token representing the compromised user, they can authenticate off the box making use of the standard Windows SSO process.

Typically, plain text credentials are found by attackers via either **Kerberoasting** or searching for unsecured plain text credentials across all accessible resources, such as network shares, Sharepoint, internal wikis, enterprise GitHub, Zendesk, etc.[8]

3. Change the cached credentials associated with their current access token to stolen credentials (e.g., legitimately via an API or "illegitimately" by *directly modifying* lsass memory)

In this scenario, rather than create a new logon session, the attacker modifies the cached credentials associated with their current access token (and hence logon session). As we shall see, many Windows **Security Support Providers** (SSPs) provide native ways to do this (and which **do not** require elevated privileges).

Alternatively, attackers can go the "direct" route and **manually** modify cached credentials stored in lsass. This requires elevated privileges in order to obtain a write handle (e.g., **PROCESS_VM_WRITE**) to lsass via **OpenProcess**. This is typical of pass-the-hash type attacks as we shall cover later on.

Access Token Manipulation attacks

This blog post will look at four common techniques used by attackers (all of which can be classified as variations of option 3 above):

- The NETONLY flag

- Pass-The-Ticket

- Pass-The-Hash

- Overpass-The-Hash

1. The NETONLY flag

```
    );
```

The key parameter to take note of here is the **dwLogonType**, which specifies the *type* of logon to perform. For example, in the case of a user physically logging into their workstation, it will be set to **LOGON32_LOGON_INTERACTIVE**. The logon type specified will determine the *type* and *privileges* of the token returned.

For example, in the case of an interactive logon, LogonUserW will return a primary access token, and, if UAC is *enabled*, this token will be a filtered token (meaning it will be medium integrity and **unelevated**). This has one exception: if the user is a local administrator account (e.g., a *-500 **SID**) Windows will automatically return an elevated token[10].

In the case of a network logon (**LOGON32_LOGON_NETWORK**), an *impersonation* token is returned (as typically this would be used by a server to perform work on the remote clients behalf). Furthermore, if the user is in the local administrators group, the token is elevated and has *all* privileges enabled[11].

These permutations of LogonUser are captured in the table below:

dwLogonType	Token returned	Cache credentials?	Is returned token elevated? (if admin)
Interactive (**LOGON32_LOGON_INTERACTIVE**)	Primary	Yes	No (UAC applies)
Interactive (Local admin account, e.g., rid-500)	Primary	Yes	Yes
Network (**LOGON32_LOGON_NETWORK**)	Impersonation	No[12]	Yes (+ *all* privileges enabled)
Network (Local admin account, e.g., rid-500	Impersonation	No	Depends on remote UAC settings[13]

Table 1 - The permutations of LogonUser for the corresponding dwLogonType

The key point is that LogonUser returns a handle to a **newly minted token**, which can now be used for impersonation.

If the token returned is a **primary** token it must first be converted in to an impersonation token via **DuplicateTokenEx** by passing a **TokenType** of TokenImpersonate[14]:

The **SetThreadToken** function can then be used to assign the returned impersonation token to the current thread:

```
BOOL SetThreadToken(
    PHANDLE Thread,
    HANDLE  Token
);
```

Alternatively, the Windows API provides the **ImpersonateLoggedOnUser** function, which will allow the calling thread to impersonate the *security context* of the user represented by the token passed:

```
BOOL ImpersonateLoggedOnUser(
    HANDLE hToken
);
```

ImpersonateLoggedOnUser has the added benefit that it will automatically check the type of the token passed and convert it to an impersonation token (via **NtDuplicateToken**) if a primary token was passed (as this token type cannot be used by a thread to impersonate)[15].

Note that from a defense evasion perspective, both these impersonation APIs are lightweight wrappers over the undocumented syscall **NtSetInformationThread** (e.g., called with a **ThreadInformationClass** of **ThreadImpersonationToken**). Therefore, they are a good target for attackers to use direct syscalls to bypass user-mode hooks via techniques such as https://github.com/jthuraisamy/SysWhispers.

Furthermore, it is important to stress that Windows has strict rules around impersonation. These are listed below and taken from the MSDN page for **ImpersonateLoggedOnUser**:

```
All impersonate functions, including ImpersonateLoggedOnUser allow the requested
impersonation if one of the following is true:
    - The requested impersonation level of the token is less than SecurityImpersonation,
such as SecurityIdentification or SecurityAnonymous
    - The caller has the SeImpersonatePrivilege privilege.
    - A process (or another process in the caller's logon session) created the token using
explicit credentials through LogonUser or LsaLogonUser function.
The authenticated identity is the same as the caller
```

Additionally, the impersonated token's integrity level must also be less or equal to the calling process's integrity level or else the impersonation call will also fail[16]. Therefore, assuming an **unelevated** attacker logs on an admin user *interactively* via stolen credentials, and UAC is enabled, they will receive an unelevated (e.g., filtered) token back and hence will have no issues impersonating the returned user and moving laterally, etc.

"The curious /NETONLY flag"[17]

An attacker may find however that attempting to log on a user with stolen credentials **fails.** This may be due to a multitude of reasons, such as the credentials are valid, but the account does not have permissions to log onto that specific workstation / they're only valid in a different domain, etc. Furthermore, the attacker may also want to avoid logging in a highly privileged account entirely, as this may appear highly anomalous in certain contexts (e.g., a domain admin logging on to a low privileged business user's host should be incredibly suspicious).[18]
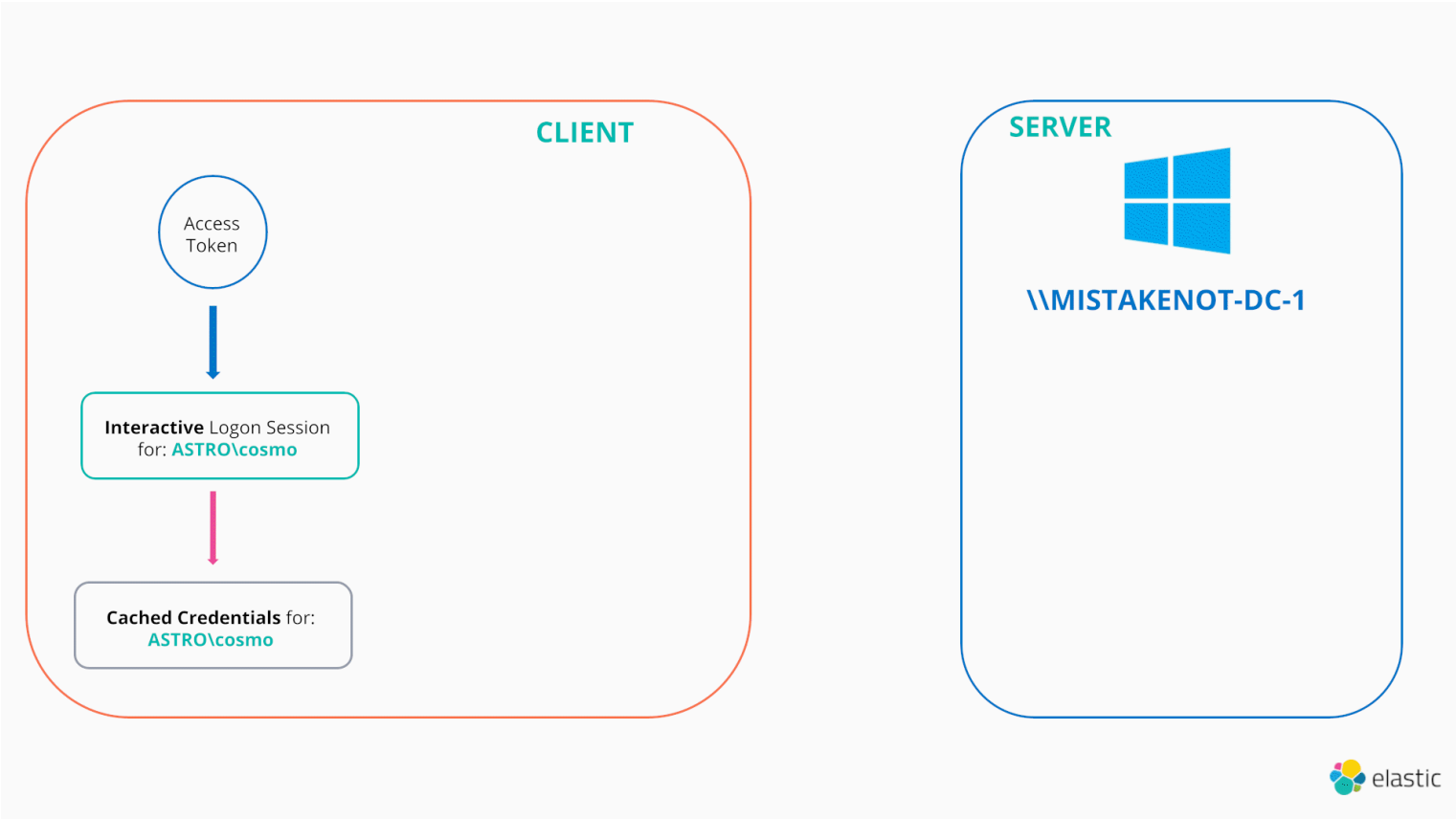
Figure 2 - How the LOGON32_LOGON_NEW_CREDENTIALS flag works under the hood

Hence, the **LOGON32_LOGON_NEW_CREDENTIALS** flag provides a native mechanism to make your current access token point to a *different* logon session and hence *different* credentials.[20]

Note, that calling LogonUserW with the LOGON32_LOGON_NEW_CREDENTIALS flag **does not** validate the credentials when the call is made (they can be complete junk), but are only validated by a Domain Controller at the time of any remote authentication requests.

As a further example, a quick review of the **code** for the 'MakeToken' task from the open source .NET C2 framework Covenant reveals exactly the same approach: it takes a username/password combination and creates a new logon session/token with them via passing the **LOGON32_LOGON_NEW_CREDENTIALS** flag before proceeding to impersonate the returned token.

Furthermore, you can replicate the exact same behaviour with **CreateProcessWithLogonW** by passing a dwLogonFlags of **LOGON_NETCREDENTIALS_ONLY**.[21]

```
BOOL CreateProcessWithLogonW(
    LPCWSTR            lpUsername,
    LPCWSTR            lpDomain,
    LPCWSTR            lpPassword,
    DWORD              dwLogonFlags,
    LPCWSTR            lpApplicationName,
    LPWSTR             lpCommandLine,
    DWORD              dwCreationFlags,
    LPVOID             lpEnvironment,
    LPCWSTR            lpCurrentDirectory,
    LPSTARTUPINFOW     lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```
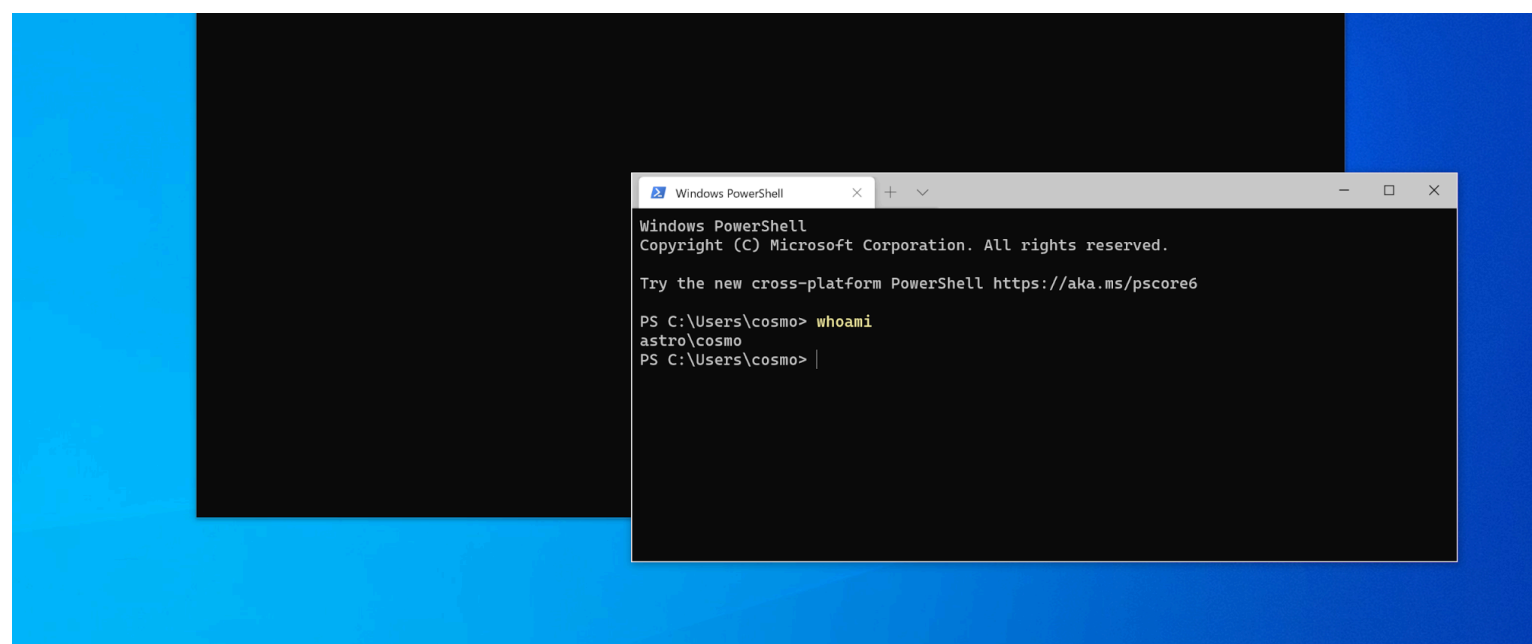
Figure 3 - Example of using the runas /NETONLY flag to spawn a new process as the user astro\cosmo but with different cached credentials.

In exactly the same way as previously described, the new command prompt appears locally to be running as the same user (i.e., the attributes cached in the token are the same for any **local** access checks; hence whoami returns 'astro\cosmo'), but any remote authentication attempts will be performed using the stolen credentials for the 'ASTRO\Administrator' user.

These logon sessions can be viewed using SysInternals' **LogonSessions** tool. Logon sessions that were created with the NewCredentials flag can be determined by the Logon type field as shown below:

Figure 4 - Example of a NewCredentials logon session which is typically generated by the NETONLY flag

Furthermore, anomalous NewCredentials logon sessions (e.g., produced via the NETONLY gadget) leave artifacts in the Windows event logs. These can be identified via the event id 4642 and a LogonType of 9. An example is shown in the image below:

```
TargetLogonId        0x1b0e53
LogonType            9
LogonProcessName     seclogo
AuthenticationPackageName Negotiate
WorkstationName      -
LogonGuid            {00000000-0000-0000-0000-000000000000}
TransmittedServices  -
LmPackageName        -
KeyLength            0
ProcessId            0x178
ProcessName          C:\Windows\System32\svchost.exe
IpAddress            ::1
IpPort               0
ImpersonationLevel   %%1833
RestrictedAdminMode  -
TargetOutboundUserName    Administrator
TargetOutboundDomainName  ASTRO
VirtualAccount       %%1843
TargetLinkedLogonId  0x0
ElevatedToken        %%1843
```
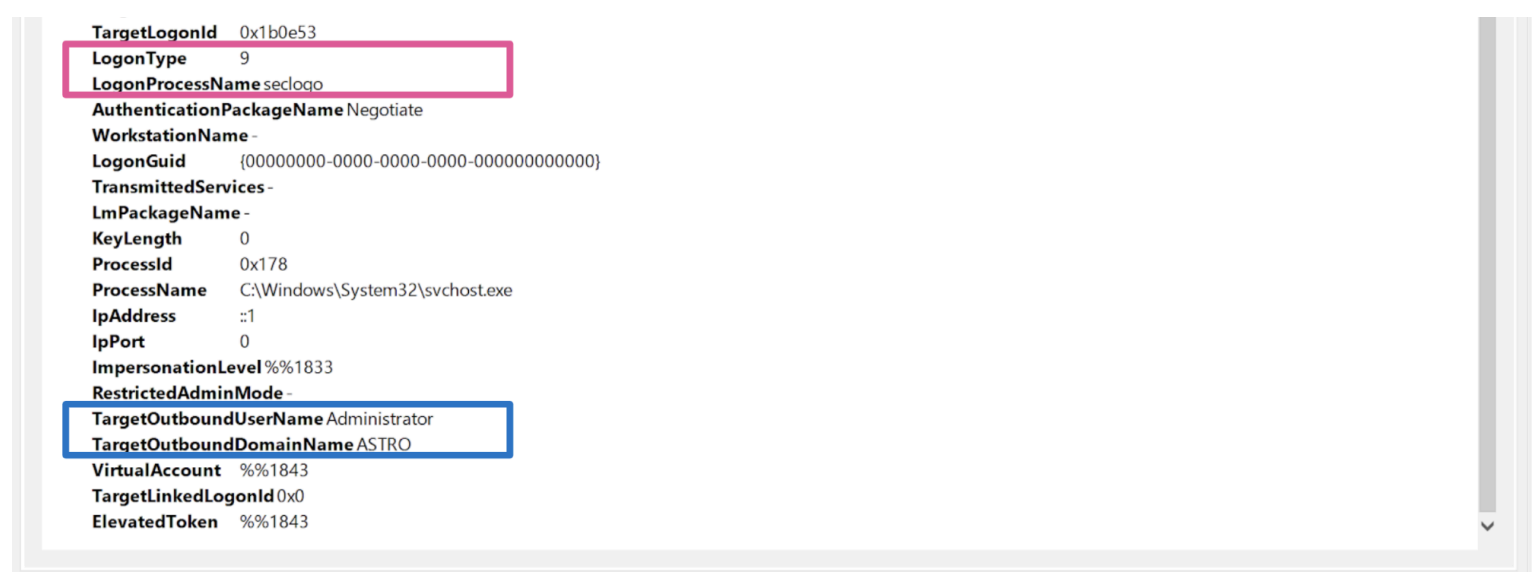
Figure 5 - Example of a Windows Event Log for Event ID 4624 which is typically generated by the NETONLY flag

Note that the original user is shown by the SubjectUserName field and the specified network only credentials (e.g., the credentials passed) are displayed in the TargetOutboundUser/DomainName fields.[22]

Auto-elevation

One further quirk from a local privilege escalation perspective is that for rid-500 accounts, **CreateProcessWithLogonW** will automatically elevate the returned token for interactive logons (e.g. it will ignore UAC). Therefore, **CreateProcessWithLogonW** can be passed a local/domain admin account in order to execute an **elevated process from a medium/unelevated context**.

This behavior can be verified using runas. For example, when runas is used to spawn a process using a local admin account (e.g., `runas /user:"Administrator" cmd.exe`), the resulting process will be elevated (e.g., high integrity). However, when a non rid-500 account is used (but which is still in the local administrators group) the resulting process will be unelevated (e.g., it will be a filtered token / medium integrity).

Notice that this behaviour is consistent with the permutations listed for LogonUserW in Table 1. Therefore, an unelevated attacker could also log on a (non rid-500) admin user as a network logon and receive an elevated token with all privileges enabled.

However, as per the impersonation rules previously outlined, the attacker should not actually be able to *do anything* with this token as any attempts to impersonate the elevated token should fail, as it has a higher integrity level than the caller. Nevertheless, it is actually possible to duplicate the elevated token, lower the integrity level of the copied token to medium (NB 'isElevated' is still true)[23], and start impersonating the elevated token from an unelevated/medium integrity context[24]. Hence, from an impersonation token perspective, you can bypass the default Windows behaviour of only elevating certain accounts and impersonate an elevated token irrespective of whether the account is a rid-500 account or not.

Process creation

Note, that by default, when you create a child process it inherits your primary token *even* if you are currently impersonating another security context[25]. For example, if you are impersonating a SYSTEM

new process.[27]

```
BOOL CreateProcessWithTokenW(
  HANDLE                hToken,
  DWORD                 dwLogonFlags,
  LPCWSTR               lpApplicationName,
  LPWSTR                lpCommandLine,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCWSTR               lpCurrentDirectory,
  LPSTARTUPINFOW        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

```
BOOL CreateProcessAsUserW(
  HANDLE                hToken,
  LPCWSTR               lpApplicationName,
  LPWSTR                lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCWSTR               lpCurrentDirectory,
  LPSTARTUPINFOW        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

For example, CreateProcessAsUserW is typically used by the operating system itself to spawn the user's shell following a successful logon (it is also used by the Secondary Logon service when a user calls creatProcessWithLogonW). In this sense, it allows a user to "inject a process into the logon session of their choice"[28]. As a note, both of these APIs are wrappers around **CreateProcessInternalW** (located in KernelBase.dll).

The key difference here is that the caller must have certain privileges to call these two APIs[29]. From an attackers perspective though the goal here is the same; obtain code execution in the security context of the target user for the purposes of moving laterally.

One interesting quirk is that the PowerShell Empire framework was forced to take this process spawning approach (which is arguably much noisier from a detection perspective) due to limitations with how PowerShell handles impersonation and multi-threading, as the notes **here** explain in more detail.

In any case, the workflow for using process spawning token manipulation techniques remains the same. Once the attacker has obtained a handle to the token (via **OpenProcess**/**OpenProcessToken** if primary token, or **OpenThread**/**OpenThreadToken** in the case of a thread impersonating) the attacker must call DuplicateTokenEx to create a local (primary) copy of the target token, and then supply this copy to either the CreateProcessWithTokenW or CreateProcessAsUserW functions.

Note that again in this case, attackers are only interested in privileged logon sessions which are **non network logins**, as network logins *do not* cache credentials and so **cannot** authenticate to other hosts.
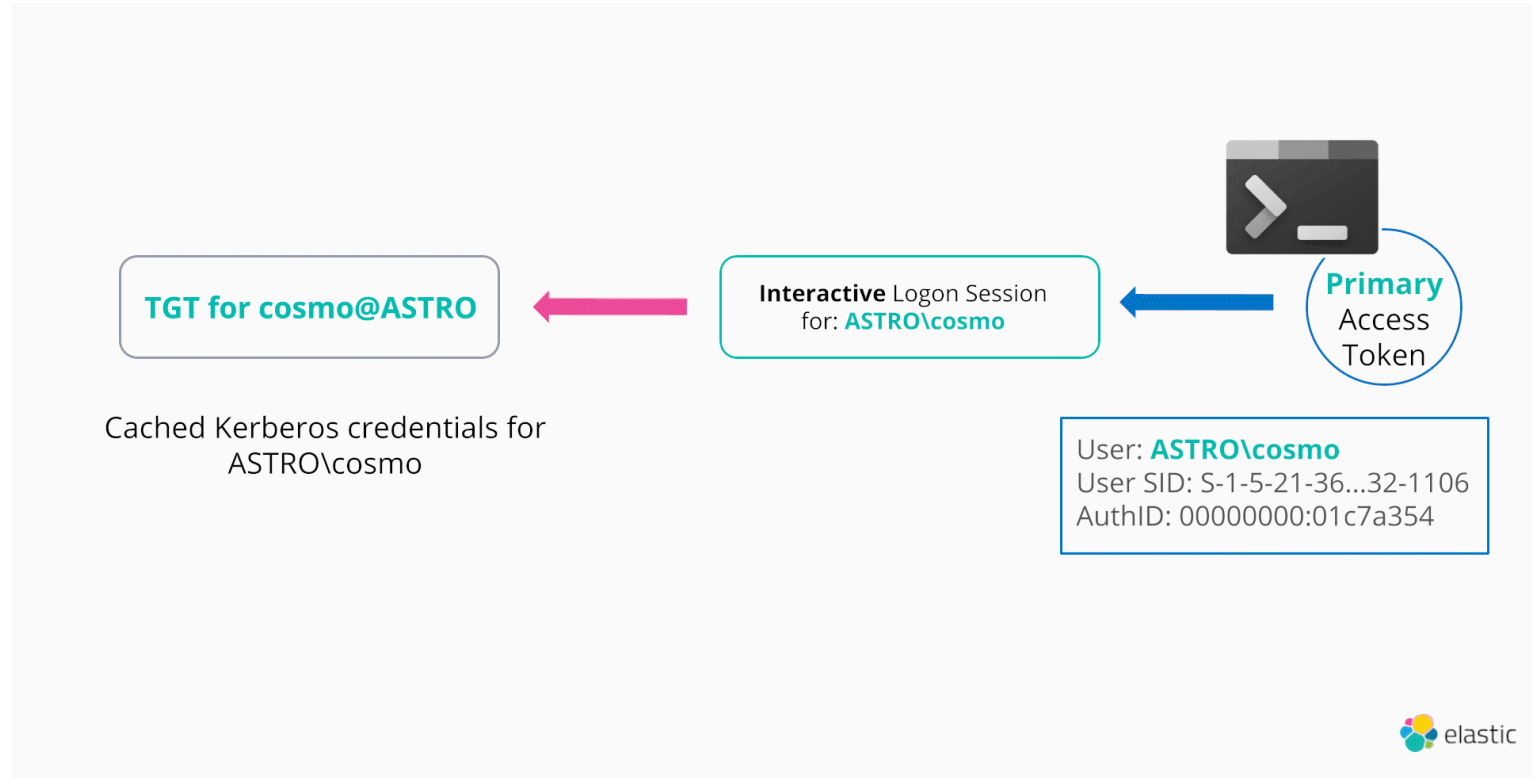
Figure 6 - How the Pass-the-ticket attack works under the hood. In this example, the user, ASTRO\cosmo, applies the stolen TGT of the ASTRO\Administrator user to their current logon session.

In order to start interacting with the Kerberos SSP and manage the Kerberos ticket cache, a process can call **LsaCallAuthenticationPackage** (located in Sspicl.dll):

```
NTSTATUS LsaCallAuthenticationPackage(
    HANDLE     LsaHandle,
    ULONG      AuthenticationPackage,
    PVOID      ProtocolSubmitBuffer,
    ULONG      SubmitBufferLength,
    PVOID      *ProtocolReturnBuffer,
    PULONG     ReturnBufferLength,
    PNTSTATUS  ProtocolStatus
);
```

Note that the user will need to have previously called **LsaConnectUntrusted** in order to obtain a connection handle to the LSA server and **LsaLookupAuthenticationPackage** to find the id of the kerberos package (MICROSOFT_KERBEROS_NAME_A). Additionally, inspection of these functions in IDA (again they can be located in Sspicl.dll) will reveal that they are connecting to the Lsa via **RPC.**[31].

Through LsaCallAuthenticationPackage, a user can make a number of sensitive requests, although the exact requests available to the user depend on whether they are elevated or not. For example, an **unelevated** user can perform basic ticket management actions[32], such as enumerating their current active tickets, purging the ticket cache, and **applying arbitrary tickets to their current logon session**[33]. Hence, this effectively enables a user to change the credentials cached with their current logon session and therefore specify **arbitrary *network only* credentials**.

Additionally, from an elevated context[34] an attacker can enumerate and dump tickets (e.g., credentials) belonging to other users, therefore providing similar functionality to mimikatz without needing to open a handle to lsass[35].

```
    ULONG KerbCredOffset;
} KERB_SUBMIT_TKT_REQUEST, *PKERB_SUBMIT_TKT_REQUEST
```

Therefore, for a KerbSubmitTicketMessage, the ProtocolSubmitBuffer parameter simply points to a block of memory consisting of a KERB_SUBMIT_TKT_REQUEST struct followed immediately by an **ASN** encoded Kerberos ticket (which is the ticket to be applied to the specified logon session). The relevant code in mimikatz for submitting KerbSubmitTicketMessage requests can be found **here** and in Rubeus **here**.

Following the call to LsaCallAuthenticationPackage, the user's TGT has now been updated to the stolen ticket. From this point forward, any attempts to access network resources by any process/thread which is linked to the user's access token/interactive logon session will **automatically** authenticate over Kerberos using the stolen TGT (e.g., by requesting different service tickets/TGS for resources across the domain).

Note, that a user can only have **one** TGT associated with their current logon session. Hence, applying a new ticket will wipe the user's previous ticket. What if an attacker would like to preserve their current TGT? In this case, once again the NETONLY flag comes to the rescue - an attacker can create a **"sacrificial" NETONLY process** via CreateProcessWithLogonW with arbitrary/junk credentials. This will create a new dummy process and, most importantly, a new logon session (and hence access token) to which a stolen TGT can be applied (and hence preserve the user's current ticket)[36].

One important conclusion to draw from this technique for defense practitioners, is that as *all* the activity is performed via LsaCallAuthenticationPackage (and hence over RPC), it does not require any **direct** interaction with lsass (N.B. direct here refers to opening a handle to lsass via OpenProcess). Furthermore, for this specific use case (ptt), all the activity is via local RPC *until* an attacker attempts to authenticate to a remote host (which will generate new logons).

As a further example, the README for **Rubeus** includes the following statement:

"Rubeus doesn't have any code to touch LSASS (and none is intended), so its functionality is limited to extracting Kerberos tickets through use of the LsaCallAuthenticationPackage() API"

Therefore, any detection logic which is predicated on handle access to lsass (e.g. via a **ObjectPreCallback** kernel routine for a specified **process** or **thread** handle operation, or a user mode hook on OpenProcess/NtOpenProcess) could miss this activity. Hence, it is a potential blind spot for, say, defenders relying on Sysmon process access events to alert on suspicious process handle access.

3. Pass-the-hash (PtH)

The last two techniques this blog will cover are examples of an attacker changing the cached credentials associated with their current access token/logon session "illegitimately" by **directly modifying** lsass memory. In the PtH scenario, the attacker's access token is unchanged and points to the same logon session, however the associated cached credentials are directly overwritten to a stolen hash. From this point, any remote authentication attempts will use the stolen hash, as demonstrated below:
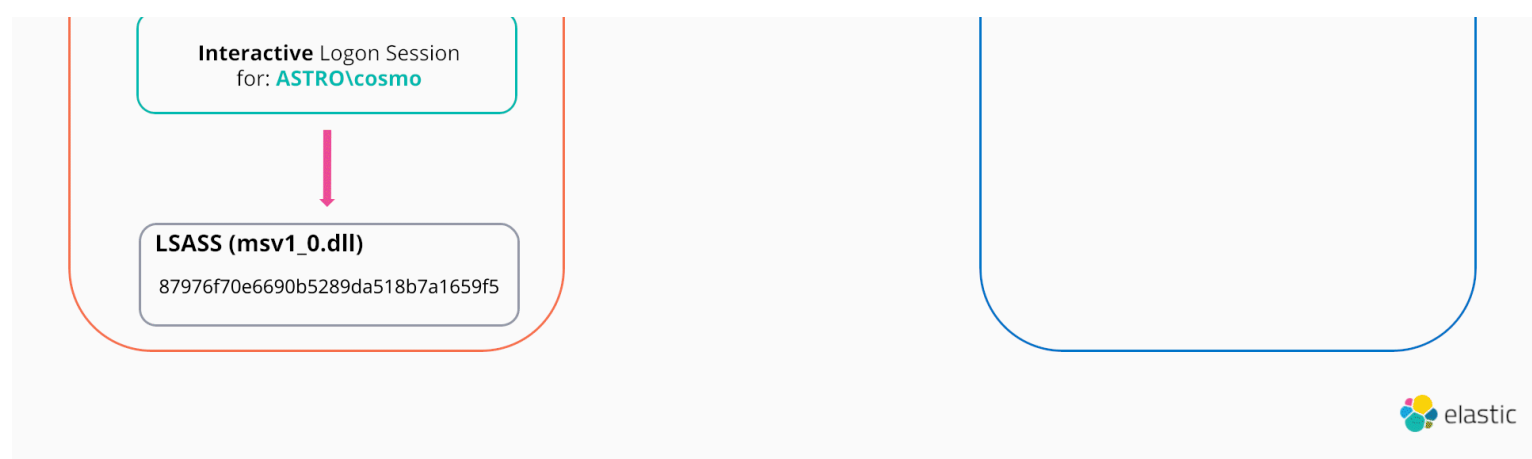
Figure 7 - How the PtH attack works under the hood. In this example, the legitimate hash of the user, ASTRO\cosmo, is overwritten in-memory with the NTLM hash belonging to the ASTRO\Administrator user.

In this sense, both PtH and OPtH can be thought as *functionally* identical to the NETONLY technique previously discussed.

The typical workflow of a PtH attack is:

- Open a write handle to lsass (e.g. via **OpenProcess**/**NtOpenProcess** with a desired access of **PROCESS_VM_WRITE**)

- Enumerate the linked list of logon sessions

- Locate the logon session of interest and identify the required authentication package (In the case of PtH/NTLM this is the **MSV1_0 authentication** package)

- Update the associated cached credentials

Note that these techniques often rely on parsing and modifying **undocumented** Windows structures. This is not something that will be covered in this blog, but more information on how this is performed can be found **here** and **here**.

Hence, once the cached credentials are updated in memory, they will **automatically** be used to authenticate remotely, as per the usual Windows SSO design, when any process/thread running as that token attempts to access a remote resource.

Note, that in this simple case, there have been **no** additional logon session / access tokens created. However, in a similar fashion to pass-the-ticket attacks, these tools will also frequently need to create new junk NETONLY processes/logon sessions in order to preserve existing credentials or to apply stolen credentials to.

As a note, in order to obtain a write handle to lsass, malware will typically take two approaches:

- Acquire SeDebugPrivilege[37]

- Steal and impersonate a SYSTEM token

The first approach was discussed in part one of this blog series, however the latter approach is a typical example of stealing/impersonating a token **for the purpose of bypassing local access checks** (e.g. **stealing a SYSTEM token** with a specific privilege enabled e.g. SeTcbPrivilege). A SYSTEM token is commonly obtained via **stealing the primary token from winlogon**.

4. Overpass-the-hash (OPtH)

It is actually performing overpass-the-hash under the hood):

```
mimikatz # sekurlsa::pth /user:Administrator /domain:ASTRO.testlab /ntlm:
c0f969f35beb20e8f09ce86ef42ccd51
```

This essentially performs the same steps as PtH, except it targets the Kerberos SSP (and hence kerberos.dll).[40]

Figure 8 - How the OPtH attack works under the hood. In this example, the legitimate hash of the user, ASTRO\cosmo, is overwritten in-memory with the hash belonging to the ASTRO\Administrator user, kicking off the normal Kerberos authentication process.

As this technique once again involves wiping the current TGT associated with the user's logon session, an attacker can use a NETONLY process (with an associated dummy logon session) to preserve their current TGT, which is exactly how mimikatz **performs** overpass-the-hash by default.

Firstly, it spawns a new process in a suspended state via CreateProcessWithLogonW with the LOGON_NETCREDENTIALS_ONLY flag. It then obtains a handle to the primary token of this suspended process and retrieves the authentication id for the *new* dummy logon session via **GetTokenInformation**. This function is used to query information cached in the token via the **TOKEN_INFORMATION_CLASS** enum, which in this case is **TokenStatistics**.

Having obtained the authentication id, mimikatz can now start enumerating the linked list of logon sessions within lsass, looking for the newly created logon session. Once it has found the target logon session (via the authentication id), it can then proceed to update the Kerberos credentials associated with it. Once the credentials are updated, the token (whose corresponding logon session is now linked to the stolen hash) can be converted to an impersonation token via DuplicateTokenEx and impersonated via SetThreadToken as we have seen previously.

Once again at this stage, any attempts an attacker makes to access resources across the network will use the domain\user and password hash combination provided as arguments to mimikatz for

authentication provider is abused, attackers act under a set of constraints that result in the same anomalous signals for access token manipulation (e.g., anomalous network only logins). These constraints are determined by the fundamental relationship between access tokens, logon sessions and cached credentials.

Ready for holistic data protection with **Elastic Security**? Try it free today, or experience our latest version on **Elasticsearch Service** on Elastic Cloud. And take advantage of our **Quick Start training** to set yourself up for success.

References

1. For a recap of how Kerberos authentication works see Programming Windows Security, Keith Brown or **https://posts.specterops.io/kerberosity-killed-the...**. Additionally, Rubeus, which is a toolkit for interacting with Kerberos, has an extremely informative **readme**, which is recommended for further reading.

2. Remember, Windows will automatically authenticate with the credentials cached in the logon session whenever a user attempts to access a network resource as per the Windows SSO mechanism. Cached credentials here can refer to any authentication provider (e.g. NTLM hashes or Kerberos tickets). NB this assumes the user is interactively logged in (non-network).

3. This is typically to avoid losing a foothold due to incident response or host isolation.

4. This is obviously only applicable to attacker activity on a compromised host, as opposed to an attacker executing code from another source e.g. remotely via **impacket**.

5. **https://clymb3r.wordpress.com/2013/11/03/powershel...**

6. See the 'steal_token' command from Cobalt Strike as an example of this technique: **https://www.cobaltstrike.com/help-beacon**

7. **This** comment from the archived PowerSploit framework should also provide further clarification on this distinction between token theft for local privilege escalation vs lateral movement.

8. Alternatively, attackers can also go the password spraying route or attempt to use NTLM sniffing/replaying attacks via tools such as **responder**.

9. Note that both LogonUserA/W are simple wrappers around **LogonUserExExW** in SspiCli.dll

10. In exactly the same way, **CreateProcessWithLogonW** can be passed a local admin account (rid-500) to execute an elevated process from a medium/unelevated context.

11. There are remote UAC registry **options** which can modify this behaviour.

12. There is an additional logon type, LOGON32_LOGON_NETWORK_CLEARTEXT, which is essentially a network logon but with cached credentials. See Programming Windows Security, Keith Brown for more information.

13. See for more info:

https://blueteamer.blogspot.com/2018/12/disabling-...
https://support.microsoft.com/en-gb/help/951016/de...

18. This is typically the main reason why option 2 is not commonly used by attackers.

19. Hence, running 'whoami' will still show the same user (as the token is still the same), despite the duplicated token having different network credentials. This is a common source of confusion when using Cobalt Strike's **make_token** command (which performs the same technique as described under the hood).

20. The Windows RPC/COM APIs also enable a user to specify network-only credentials. For example, this can be achieved for RPC by calling **RpcBindingSetAuthInfoExW** and passing a SEC_WINNT_AUTH_IDENTITY structure via the AuthIdentity parameter. For more information see Programming Windows Security, Keith Brown and **https://docs.microsoft.com/en-us/windows/win32/wmisdk/setting-authentication-using-c-**.

21. While the two flags have different names, their meaning is the same; these credentials are only to be used on the network.

22. Note there are still **ways** around creating suspicious event logs for anomalous logon sessions.

23. This is a James Forshaw trick - see the following blog for more detail: **https://www.tiraniddo.dev/2017/05/reading-your-way...**. Additionally, **TokenViewer** is an excellent tool for experimenting with this type of technique.

24. With this resulting impersonation token it is possible to write a file to System32 etc.

25. There still may be legitimate reasons for impersonating prior to calling an API though, such as to obtain a privilege you don't currently have before calling an API which requires it (although note some APIs do automatically enable privileges).

26. There are a few ways around this. For example, you can spawn a process as the child of a SYSTEM process by obtaining a handle to a SYSTEM process via OpenProcess with the **PROCESS_CREATE_PROCESS** access right. This HANDLE can then be passed to **NtCreateProcess** as the ParentProcess parameter. This can also be achieved via the PROC_THREAD_ATTRIBUTE_PARENT_PROCESS parameter and CreateProcess: **https://gist.github.com/xpn/a057a26ec81e736518ee50...**

27. Bizarrely, CreateProcessWithTokenW takes a dwLogonFlags argument despite also requiring a handle to an existing token, which by definition, should already have a corresponding logon session. It seems likely that this is something to do with loading the user profile.

28. Programming Windows Security, Keith Brown

29. Specifically, SE_IMPERSONATE_NAME for CreateProcessWithTokenW and SE_INCREASE_QUOTA_NAME (&) SE_ASSIGNPRIMARYTOKEN_NAME (if token is not assignable) for CreateProcessAsUserW

30. A recap of Kerberos authentication can be found **here** and see the following for more information on kerberos related attacks: **https://www.blackhat.com/docs/us-14/materials/us-1...**,

,
https://github.com/GhostPack/Rubeus#readme

31. **https://googleprojectzero.blogspot.com/2019/12/cal...**

32. E.g. the native Windows tool **klist** offers similar functionality and is clearly a wrapper around LsaCallAuthenticationPackage.

33. Note that an unelevated user can only apply tickets to their own logon session; elevated privileges are needed to apply a TGT to a different logon session.

34. There are some caveats/subtleties to this statement which are better answered by the Rubeus **readme**. In short though, the caller needs to register an LSA connection via **LsaRegisterLsaProcess** which requires the SeTcbPrivilege privilege (i.e. the caller is part of the trusted computing base).

35. As an observation, you can also talk to the msv1_0 authentication package via LsaCallAuthenticationPackage and send the following message types: **https://docs.microsoft.com/en-us/windows/win32/api...**, although I have not investigated whether it is also possible to retrieve NTLM credentials through this interface.

36. For more information see the Rubeus github repository **readme**, which has a fantastic write up of lots of kerberos related functionality and opsec considerations.

37. See **here** for an example of enabling a privilege

38. This can be verified by looking at PsOpenProcess/Thread in IDA and looking for a call to SePrivilegeCheck.

39. Note, that acquiring SeDebugPrivilege tends to be very noisy from a detection logic perspective.

40. Note the hash/key can be rc4_hmac (e.g. NTLM), aes128_hmac, aes256_hmac etc.. see **here** for more.

41. See for more detail: **https://www.blackhat.com/docs/us-14/materials/us-1...**

42. As a note, Rubeus' **asktgt** functionality performs a variant of overpass-the-hash via building raw AS-REQ traffic for a given hash from an unelevated context and without needing to touch lsass.

NOUS SUIVRE

Blog

Newsroom

EMPLOIS

Carrières

Portail dédié aux offres d'emploi

RELATIONS INVESTISSEURS

Ressources pour investisseurs

Gouvernance

Finances

Bourse

EXCELLENCE AWARDS

Lauréats précédents

ElasticON Tour

Devenir sponsor

Tous les événements

Devenir partenaire

CONFIANCE ET SÉCURITÉ

Centre de confiance

Portail EthicsPoint

Rapport ECCN

E-mail du service déontologie