Exploring Token Members Part 1

jsecurity101.medium.com/exploring-token-members-part-1-48bce8004c6a

January 4, 2022

Jonathan Johnson

Jan 4

·

6 min read

·

Introduction

In an attempt to understand access tokens at a deeper level as of late, I have come across a couple of members within the **TOKEN** structure that have connected some dots for me. They are not novel findings, but I hope these findings help someone else, as they have me. This write-up does assume a small amount of knowledge on access tokens, but I will try to do a quick TLDR.

For those that are not aware, access tokens are a kernel object (**nt!_TOKEN**) that contains various members that serve to identify the security context (user security identifier, security identifier, group memberships, and privileges) of a process or thread. Unless a token is explicitly assigned to a thread, all threads will inherit the token of the primary thread (i.e., the first thread started in a process), which is also known as the primary token. All actions the process takes will fall under the security context of that token.

Every token is tied to a logon session. Anytime a user logs in, a logon session is created and a token is tied to that session. I had a couple of questions about this:

1. How could I find the access token that was created upon logon?
2. How is the logic between linked tokens handled?

Luckily when searching for these answers I came across a member within the **TOKEN** structure, called: **LogonSession**. This member is backed by another structure: which held all the answers to my questions.

_SEP_LSA_LOGON_REFERENCE

My current understanding is that thestructure holds information about a particular logon session. If you pull via within WinDbg, the return value is a pointer to this structure. This structure holds some interesting members:

```
 +0x000 Next : Ptr64 _SEP_LOGON_SESSION_REFERENCES +0x008 LogonId : _LUID +0x010
BuddyLogonId : _LUID +0x018 ReferenceCount : Int8B +0x020 Flags : Uint4B +0x028
pDeviceMap : Ptr64 _DEVICE_MAP +0x030 Token : Ptr64 Void +0x038 AccountName :
_UNICODE_STRING +0x048 AuthorityName : _UNICODE_STRING +0x058 CachedHandlesTable :
_SEP_CACHED_HANDLES_TABLE +0x068 SharedDataLock : _EX_PUSH_LOCK +0x070
SharedClaimAttributes : Ptr64 _AUTHZBASEP_CLAIM_ATTRIBUTES_COLLECTION +0x078
SharedSidValues : Ptr64 _SEP_SID_VALUES_BLOCK +0x080 RevocationBlock :
_OB_HANDLE_REVOCATION_BLOCK +0x0a0 ServerSilo : Ptr64 _EJOB +0x0a8 SiblingAuthId :
_LUID +0x0b0 TokenList : _LIST_ENTRY
```

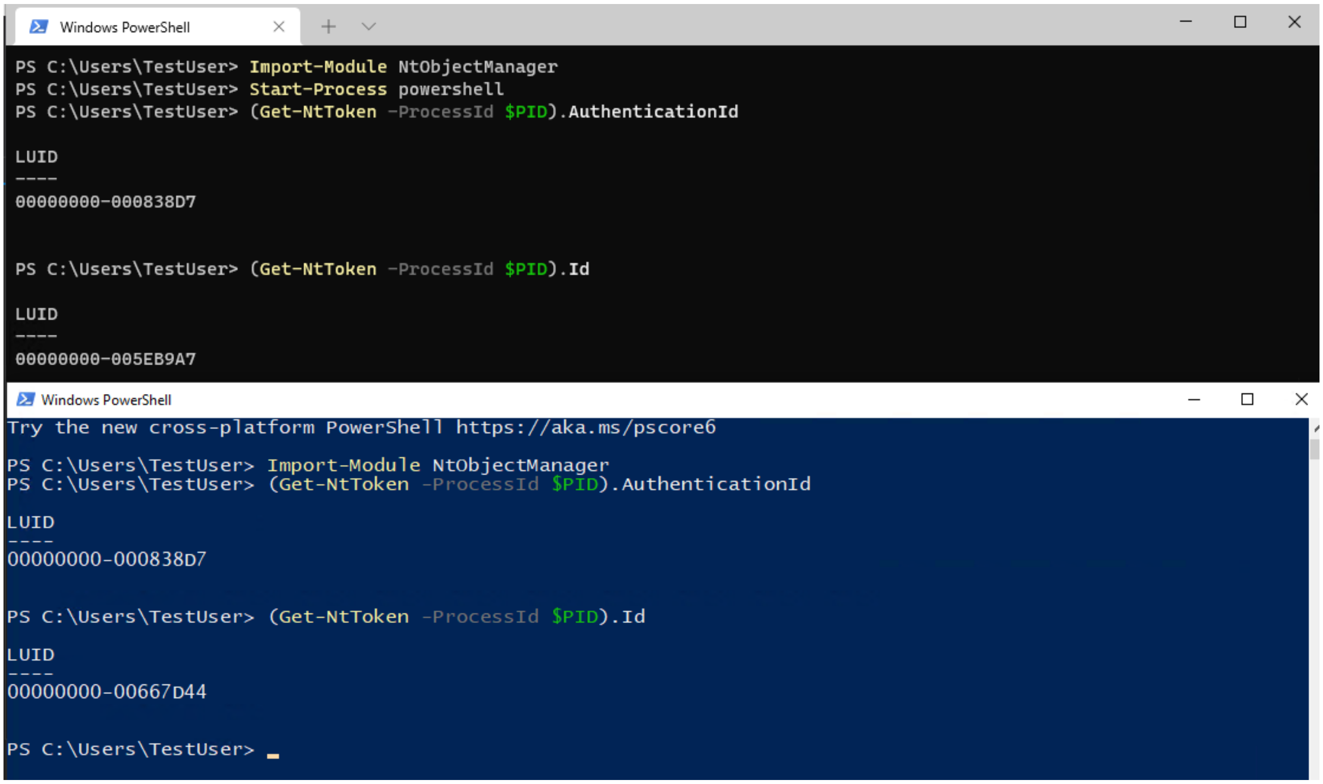
The first member that stands out to me is — **Token**.

Original Token

Whenever a logon session is successful, an access token is generated (lets call this token 1) to create the initial processes for that user's session (See Windows Internals Part 1, Chapter 2 for more). Knowing that and then knowing that when new processes are created, the child duplicates the parent process's token — I was curious if the kernel somehow kept track of token 1 somewhere.

Within thestructure there is a member called **Token** that caught my eye. This member is a pointer to another **TOKEN** structure. After some digging, I was able to confirm that this was the original kernel token object created upon that user's successful logon. However; let me show how I went about proving that:

First, I have two processes. One is the parent of the other.



As seen above, by using NtObjectManager from James Forshaw I was able to pull the logon ids for each processes token via the token member — **AuthenticationId**. That value was: **00000000–000838D7**.

Next, I was able to pull each token's id, a member used to identify different token objects. These two values were different and so were the pointer values within WinDbg when pulled from the **EPROCESS** structure, so for now that is enough proof that the child process duplicates the parent primary token and applies it to its process (although — I hope to show this more in-depth in a future post).

Lastly, I went into WinDbg and pulled the pointer value of the token object out of each process and looked to see if the **LogonSession.Token** members were equal.

Process 1:

```
Searching for Process with Cid == d30PROCESS ffff9e0f62fda080 Image: powershell.exe
Token ffffd7834ebf0770 +0x0d8 LogonSession : 0xffffd783`47e53c70
_SEP_LOGON_SESSION_REFERENCES +0x030 Token : 0xffffd783`47fe4770 Void
```

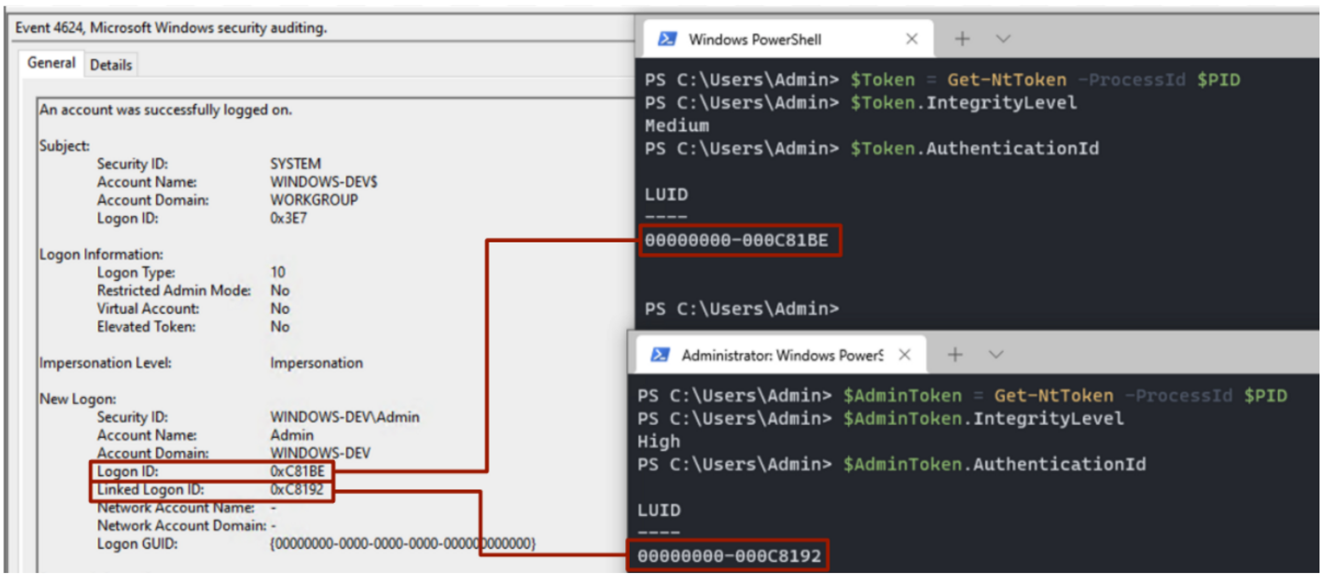
Process 2:

```
Searching  Process with Cid == 136cPROCESS ffff9e0f62ed5080 Image: powershell.exe
Token ffffd7834f407060 +0x0d8 LogonSession : 0xffffd783`47e53c70
_SEP_LOGON_SESSION_REFERENCES +0x030 Token : 0xffffd783`47fe4770 Void
```

Above we can see that two separate processes running under the same security context have two separate tokens but when the token's logon sessions are pulled, they both have the same original token. Again, this is the original token object created upon that user's successful logon session. I pulled that token's **LogonSession.Token** information and equaled that token value as well.

Linked Tokens/Logon Sessions

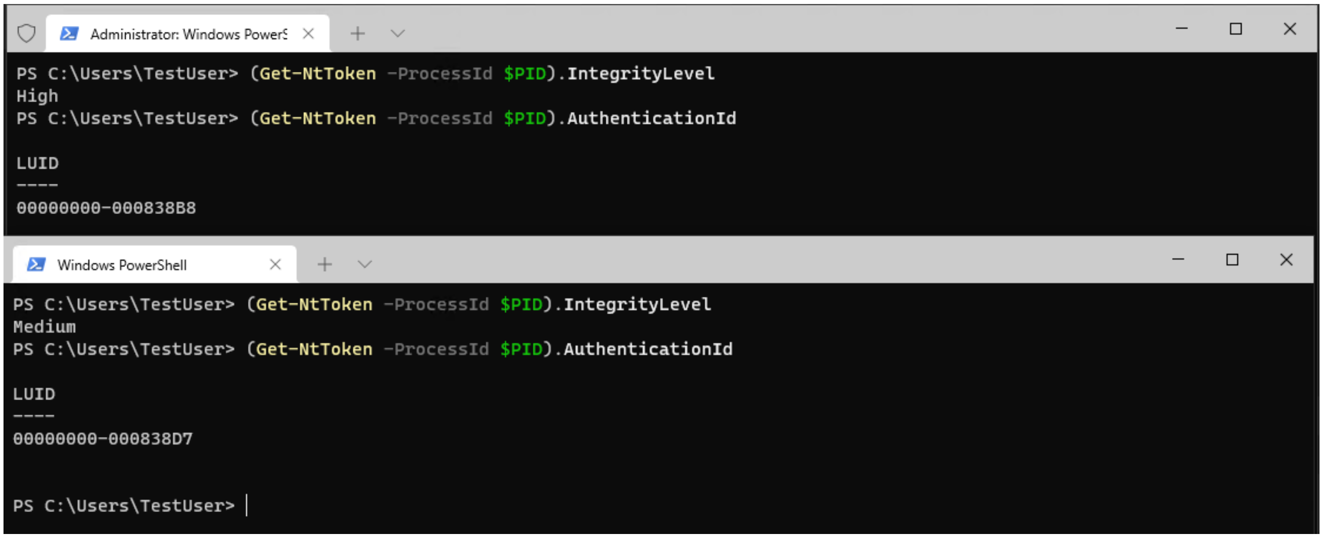
Linked tokens or sometimes referred to as "split tokens" occur when an administrator or a user that has been granted a sensitive privilege logins in. Two authentication requests are made, resulting in two separate logon sessions. One for the non-elevated token, another for the elevated token. For a touch upon this information and why this occurs, please see my last post: "Better Know a Data Source": Process Integrity Level.



I've always wanted to dive into this process more, however. Say I have a Powershell prompt and I run , how does the OS know how to transition from the non-elevated token into the elevated token (with a UAC prompt between the actions — I will not be covering UAC internals).

Turns out — that within the structure there is a member called **LogonId** and **BuddyLogonId**. As suspected, the LogonId member holds the LogonId of the current session. The **BuddyLogonId** however holds the **LogonId** of the linked session.

```
BuddyLogonId +0x008 LogonId : _LUID +0x010 BuddyLogonId : _LUID +0x000 LowPart :
0x838d7 +0x000 LowPart : 0x838b8
```



A further step could be taken to correlate these logon sessions via in WinDbg, then track down its token. This makes sense (from a high level) now that it's possible when that transition happens this value is queried to see if a BuddyLogonId exists to allow that elevated request or not.

Bonus: Originating Logon Session

The last thing I would like to show is how to identify when the logon session is responsible for another logon session.

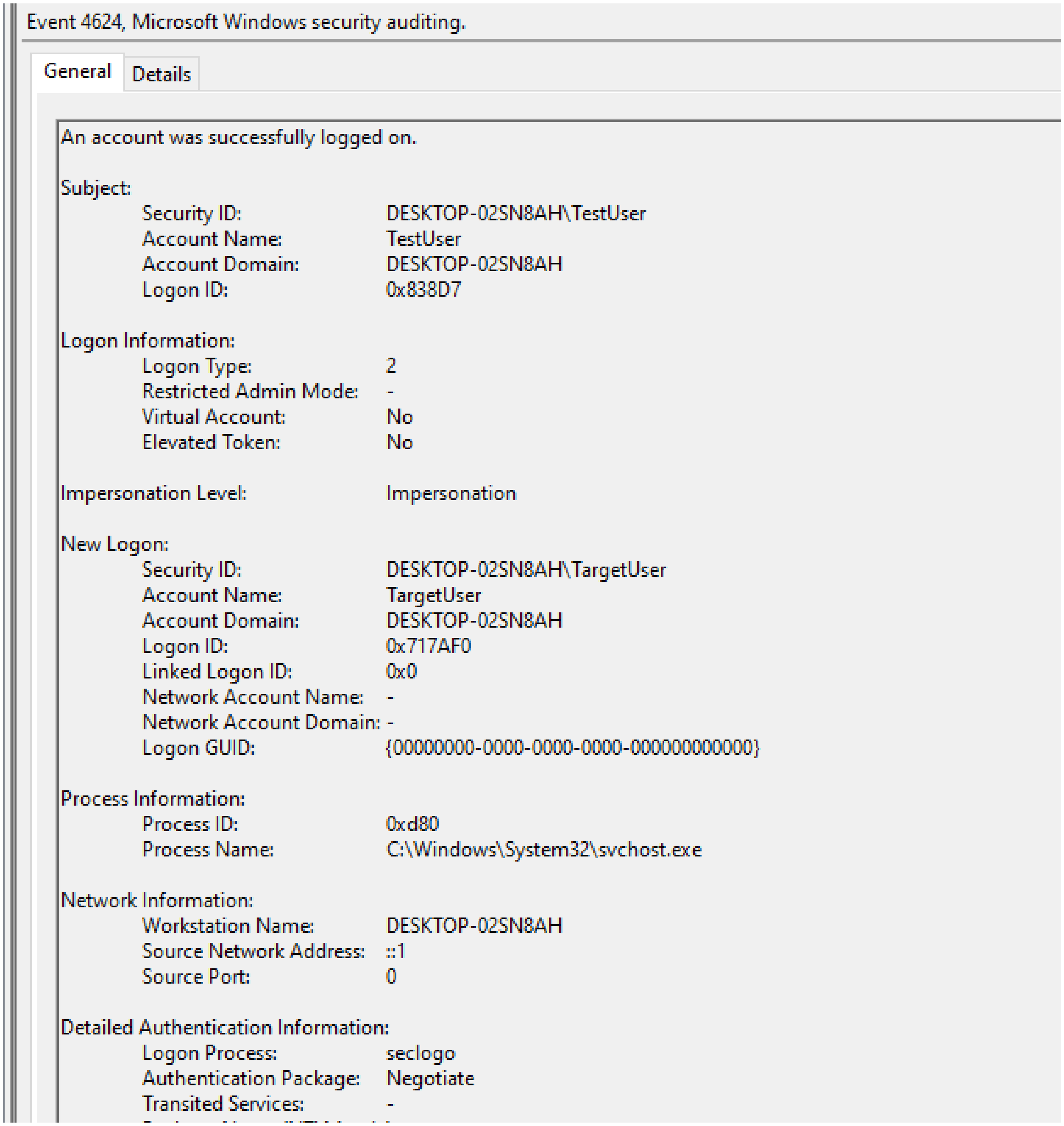
Scenario:

User logs on a new user to use powershell via RUNAS.

Command:

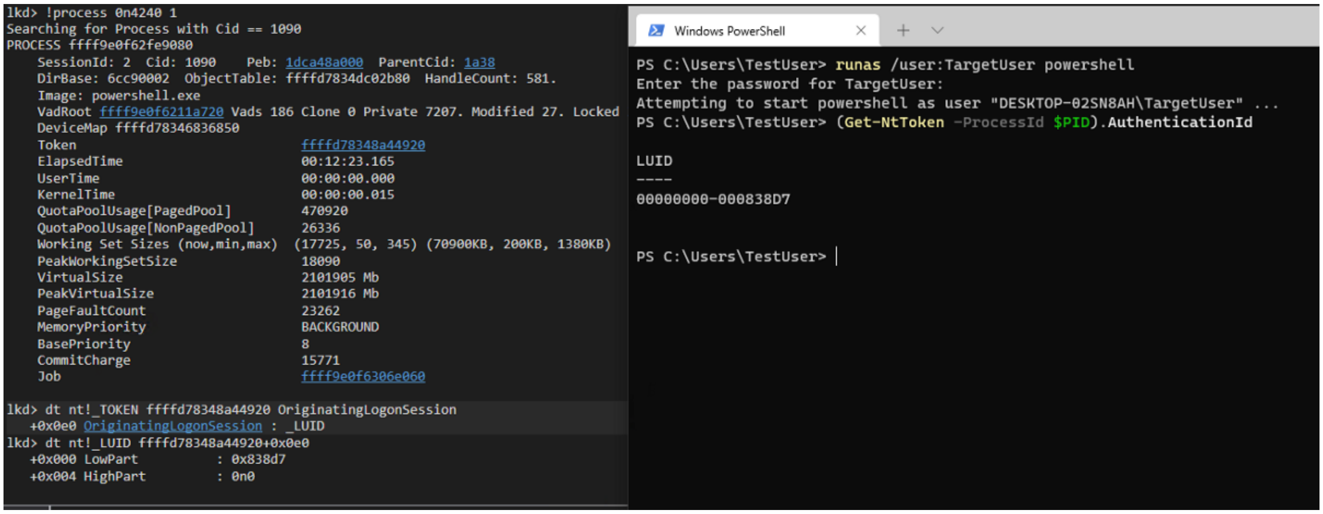
```
runas /user:TargetUser powershell
```

This result in a logon session being created, which can be seen within Windows Security Event: 4624:



The attribute in this log I want to focus on is the **SubjectLogonId**. It can be seen that TestUser was responsible for the logon and it pulled TestUser's LogonId, but is that information stored within a TOKEN's structure? Yes! There is a member called **TOKEN.OriginatingLogonSession** will show this information.

If I were to pull the token for that new process via WinDbg, then look at LUID value stored in the **TOKEN.OriginatingLogonSession** member, I will be able to correlate those two values:



Conclusion

As I go through my research I like to showcase things that I find, but most importantly the process I followed to acquire those findings as a guide or reference. The things I shared are not anything novel by any means, but I hope this can serve as a reference someday to accelerate someone's research. As I continue to go through more token research, I hope to share more.

References

Thank you to both and for confirming these findings, but also for taking the time to teach me more on the way.