Exploring Token Members Part 2

jsecurity101.medium.com/exploring-token-members-part-2-2a09d13cbb3

February 16, 2022

Jonathan Johnson
Feb 16

·

9 min read

·

Introduction

The Elastic Research team recently released work surrounding stripping the Windows Defender binary (MsMpEng.exe) of its privileges, making it effectively useless. Being that MsMpEng.exe runs under a Protected Process Light Process (PPL) state, this work led to some research of my own particularly surrounding the ability/inability of impersonating the token of a PPL process.

> **Note:** *I won't be going over the internals of PP/PPL process as I think the Elastic blog did a great job at this and as James Forshaw and Alex Ionescu hold work surrounding this protection model. I will link those in the reference section below.*

Original Thoughts

Once the work from Elastic came out I had some great conversations with various people surrounding tokens and one question came up — Being that a SYSTEM IL process has the token rights to strip the MsMpEng's token rights could it not just impersonate that token instead and assume PPL?

My initial answer to this question turned out to be too simplified and there was more going on behind the scenes than I originally thought. Originally I thought one could impersonate a PPL process (given they were already assuming SYSTEM), but since the Protection field was tied to the EPROCESS structure and not to the token, not allowing the PPL state to translate to the token. In turn, only returning back SYSTEM to the one trying to impersonate and not the level of protection of the target token.

> **Note:** *The EPROCESS structure holds a Protection member that holds information about if a process is protected, if so what protection level is a combination of the protection type and the protection signer. This is explained well in by Alex Ionescu.*

After a conversation with Alex Ionescu (thank you Alex), he was kind enough to let me know that it was a bit more complicated than that and told me to look into Token Trust SIDs and other checks that are being made. This caught me by surprise as I originally thought that the Protection member was set in the EPROCESS and was used explicitly to prevent the access of non-PPL processes from accessing it and performing things like termination. However, it seems there are times where resources, like **\KnownDLLs**, that further restricts access based on the TrustLevel ACE. Let's dive in a bit more.

TokenTrustLevel & Trust ACEs

Although I originally couldn't find any Microsoft documentation on this token member, I did find some very helpful information in a write-up on CVE-2018–8134 on Exploit-DB by James Forshaw and presentation slides from a talk that Alex Ionescu and James Forshaw put together — Unknown Known DLLs. I later found that in the Windows Internals Part 1 Chapter 7 there was a section regarding Trust ACEs. After this material, I was able to understand 2 things about this.

1. An entry specifying a trust level is set to an ACE to require a level of access before allowing a certain operation to be done.
2. The Token Trust SID (also referred to as the TrustLevelSid) holds a value that represents its level of access. This value is set during the function, which calls to set the Token TrustLevel. To my knowledge, there is no way to set a token's TrustLevel in user-mode, but it can be retrieved via .

After reading this information I had three questions:

1. What was the actual member name of this value in the TOKEN structure and what were possible return values?
2. Where was this ACE value applied?
3. Does James' exploit still work?

What was the actual member name of this in the TOKEN structure and what were possible return values?

To tackle the first question I took a look at the MsMpEng.exe as we know that is a protected level process within WinDbg. After a quick glance at the token structure, I was able to identify that the member — TrustLevelSid was the member I was looking for. I then pulled MsMpEng's token and this TrustLevelSid value in Windbg:

```
dt nt!_TOKEN ffff8f83aa2050a0 TrustLevelSid +0x450 TrustLevelSid : 0xffff868d`7d58af0
Void
```

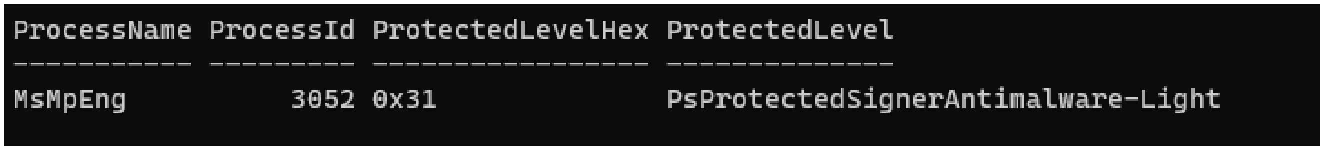
This looks like a pointer to a SID value to me, so I just called `!sid` in WinDbg with this pointer value and was able to get a result:

```
!sid 0xffff8f83a95a3b10SID is: S-1–19–512–1536
```

The trust SID can be seen with `!token` as well

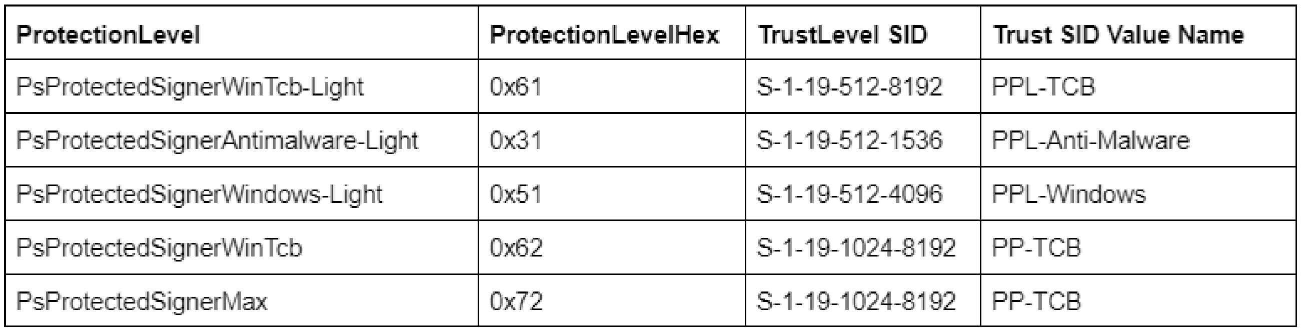
```
!token ffff8f83aa2050a0_TOKEN 0xffff8f83aa2050a0...Process Token TrustLevelSid: S-1-
19-512-1536
```

According to Alex and James these values specify what level of protected access or trust this token has. In their talk — Unknown Known DLLs they provided the different SID values and the corresponding access the SID values relate to. Highly suggest checking that out, but comparing this value to their mapping it seems MsMpEng's token has the access "PPL — AntiMalware". This seemed to be the same value given to the Protection flag for this process as well.



```
ProcessName ProcessId ProtectedLevelHex ProtectedLevel
----------- --------- ----------------- --------------
MsMpEng          3052 0x31              PsProtectedSignerAntimalware-Light
```

After discovering this I then wanted to pull the values that Alex and James provided and correlate them with a small sample set of protected processes to see what the values were from the process level to the token trust level.



ProtectionLevel	ProtectionLevelHex	TrustLevel SID	Trust SID Value Name
PsProtectedSignerWinTcb-Light	0x61	S-1-19-512-8192	PPL-TCB
PsProtectedSignerAntimalware-Light	0x31	S-1-19-512-1536	PPL-Anti-Malware
PsProtectedSignerWindows-Light	0x51	S-1-19-512-4096	PPL-Windows
PsProtectedSignerWinTcb	0x62	S-1-19-1024-8192	PP-TCB
PsProtectedSignerMax	0x72	S-1-19-1024-8192	PP-TCB

This is great, I am able to identify the levels of access a token may hold but what about from a resource perspective?

Where was this ACE value applied?

After learning about \KnownDLLs I used it as an example to find this ACE value and where it was stored. I was able to do this via NtObjectManager:

```
PS> $KnownDll = Get-NtDirectory PS> $KnownDll.SecurityDescriptorOwner DACL ACE Count
SACL ACE Count Integrity Level — — — — — — — — — — — — — — — — — — — — — — — — — — — -
BUILTIN\Administrators 5 1 NONE
```

Being that typically SACLs are used for logging and auditing and that DACLs are used to determine access to an object, my assumption was that this ACE would be set within the DACL:

```
PS > $KnownDll.SecurityDescriptor.DaclType User Flags Mask — — — — — — — — — — Allowed
BUILTIN\Administrators None 000F000FAllowed Everyone None 00020003Allowed APPLICATION
PACKAGE AUTHORITY\ALL APPLICATION PACKAGES None 00020003Allowed NT
AUTHORITY\RESTRICTED None 00020003Allowed APPLICATION PACKAGE AUTHORITY\ALL
RESTRICTED APPLICATION PACKAGES None 00020003
```

However…it wasn't. Weird. I saw there was a SACL set, so I checked that.

```
PS > $KnownDll.SecurityDescriptor.SaclType User Flags Mask — — — — — — — — — —
ProcessTrustLabel TRUST LEVEL\ProtectedLight-WinTcb None 00020003
```

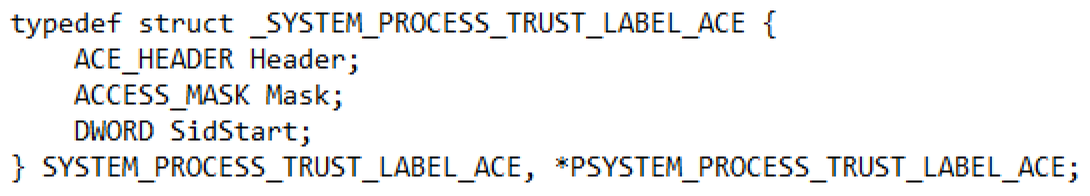
Sure enough…there it is. This was extremely odd to me as I thought that SACLs were only used for logging and auditing. I couldn't find any information on access checks being set to the SACL like this before. So I reached out to James and he was kind enough to let me know that back in Vista when Microsoft was implementing mandatory labels they made a design decision that whenever they want to implement some special ACE (non-discretionary ACE) they stick it in the SACL to avoid breaking code/access previously set.

A mandatory label ACE can be seen if the SACL is pulled for MsMpEng.exe:

```
$MsMpEngToken.SecurityDescriptor.SaclType User Flags Mask — — — — — — — — — —
MandatoryLabel Mandatory Label\System Mandatory Level None 00000001
```

I then found a Win32 API that adds a mandatory ACE to a SACL — AddMandatoryAce. Not to dive into this too much, but it seems this is one way to set mandatory ACEs.

Back to the TrustLevel ACE. This ACE structure can be found in `winnt.h` within the Windows SDK:



```
typedef struct _SYSTEM_PROCESS_TRUST_LABEL_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} SYSTEM_PROCESS_TRUST_LABEL_ACE, *PSYSTEM_PROCESS_TRUST_LABEL_ACE;
```

Process TrustLabelACE structure definition

Great, now I know where this value is stored but what does the TrustLevel ACE restrict? When looking DACL (shown above), it would seem that Administrators have some pretty powerful access.

```
PS C:\> Get-NtAccessMask -AccessMask 0x000F000F -ToSpecificAccess DirectoryQuery,
Traverse, CreateObject, CreateSubDirectory, Delete ReadControl, WriteDac, WriteOwner
```

However; when I try to create an object I get access denied:

```
PS C:\> $dll = New-NtSection \KnownDlls\pwnd.dll -Size 4096New-NtSection :
(0xC0000022) — {Access Denied}A process has requested access to an object, but has
not been granted those access rights.
```

Odd. I thought I had the correct rights to create the object. Let's take a look at the rights within the TrustLevel ACE:

```
PS C:\> Get-NtAccessMask -AccessMask 0x00020003 -ToSpecificAccess DirectoryQuery,
Traverse, ReadControl
```

Ah. What seems to be happening is, this ACE is only going to allow access higher than **0x00020003** if the token attempting to access it holds a TrustLevelSid of **ProtectedLight-WinTcb/S-1–19–512–8192** or higher. I was able to confirm this by going into WinDbg and changing a token TrustLevelSid value to match that. Afterward, I was able to create an object within **\KnownDlls** just fine. Alex was kind enough to let me know that this type of mask is referred to as a "constraint" mask. Whereas other masks — say in a DACL have an access mask. An access mask has a value that states the explicit rights a user/group might have to an object. This constraint will not allow access past a specific value without holding a trust value. In this case, no one can go past **Query/Read** — **0x00020003** without holding a TokenTrust Sid of S-1–19–512–8192 or higher.

Does James' exploit still work?

Short answer — no. This exploit was built on when a token with a TrustLevel is duplicated, the TrustLevel stays with that token. That is fine because the token wasn't used yet. The TrustLevel should be cleared when the token is used. However; that wasn't being done and he was able to leverage that duplicated token and set the token to another process via NtSetInformationProcess, which then he was able to write to **\KnownDlls**. I confirmed through NtObjectManger that you can still duplicate a token with a TrustLevel and it stays intact, but when attempting to use it via impersonation or setting it to another process it fails.

Circling Back

The original goal of this blog was to answer the question — can you impersonate the token of a PPL process. The answer is — yes, but not in the way someone might want. Being that a protected process is meant to reduce access to its process/resources a typical administrator can't access it with enough rights to "matter". Again, using NtObjectManager:

```
PS > $MsMpEng = Get-NtProcess -ProcessId 3056 -Access QueryLimitedInformationPS>
$MsMpEngToken.GrantedAccessQuery
```

Hence why the Elastic team impersonated a SYSTEM level process before stripping the Defender binary of any privileges it has. So let's assume SYSTEM moving forward.

```
PS > $MsMpEng = Get-NtProcess -ProcessId 3056 -Access QueryLimitedInformationPS>
$MsMpEngToken.GrantedAccessAssignPrimary, Duplicate, Impersonate, Query, QuerySource,
AdjustPrivileges, AdjustGroups, AdjustDefault, AdjustSessionId, Delete, ReadControl,
WriteDac, WriteOwner
```

Theoretically, based on this output the calling process has the correct token rights to impersonate the MsMpEng token. However; remember James' exploit that is now fixed? This token is cleared after it is used. So although I can impersonate it, I should only get a SYSTEM level token back….that was already obtained. James shows how to prove this within his and Alex's talk, but I also wrote some code to test it and that trust level won't be transferred.

Conclusion

This blog was meant to be a brain dump spiced up with a little methodology of how I was able to answer a question I had after seeing the Elastic research team.

Again, a big thank you to James and Alex for being willing to answer a lot of my questions. This work was just a continuation and reiteration of their work.

References