nt!_SEP_TOKEN_PRIVILEGES - Single Write EoP Protect

Kyriakos 'kyREcon' Economou

www.anti-reversing.com

TL;DR: Abusing enabled token privileges through a kernel exploit to gain EoP it won't be enough anymore as from **NT kernel version 10.0.15063** are 'checked' against the privileges present in the token of the calling process. So you will need two writes.

And now the long form...

While developping a kernel exploit that affects the products of a couple of AV vendors, I had to make changes in it after each major update of Windows 10.

The bug allows to control a kernel pointer dereference which leads to full EIP control. This is enough by design to easily exploit this situation in Windows versions previous to Windows 8.0. That is because from Windows 8.0 onwards, the kernel took advantage of the SMEP CPU feature which basically will BSOD the host if you attempt to execute in kernel-mode an instruction that resides in userland.

That being said, I had to turn the bug from direct EIP control into a Read-Write Primitive.

First choice and easy win was to set the *nt!_OBJECT_HEADER.SecurityDescriptor* pointer of an elevated process to NULL and inject a remote thread in it. This will work fine up to **Windows 10 v1511**.

First big change came with the release of **NT kernel version 10.0.14393 (v1607)** where the aforementioned attack was (finally) mitigated. You can read more in this post that I had written for Nettitude Labs. There is also a local copy here in pdf format.

So, since turning an elevated process into a sitting duck didn't work anymore, I had to use another way. Well, I guess when you manage to get a Read-Write primitive the entire host becomes a sitting duck, but let's leave this discussion for the pub.

Next easy win was, of course, to instead of turning a giant into an ant, why not turn an ant into a giant. In other words, by enabling specific privileges into our process we can then interact with elevated processes at will.

To make this work we can instead abuse *nt!_SEP_TOKEN_PRIVILEGES* structure which can be found as a member of the nt!_TOKEN structure.

```
dt nt!_SEP_TOKEN_PRIVILEGES
   +0x000 Present          : Uint8B
   +0x008 Enabled          : Uint8B
   +0x010 EnabledByDefault : Uint8B
```

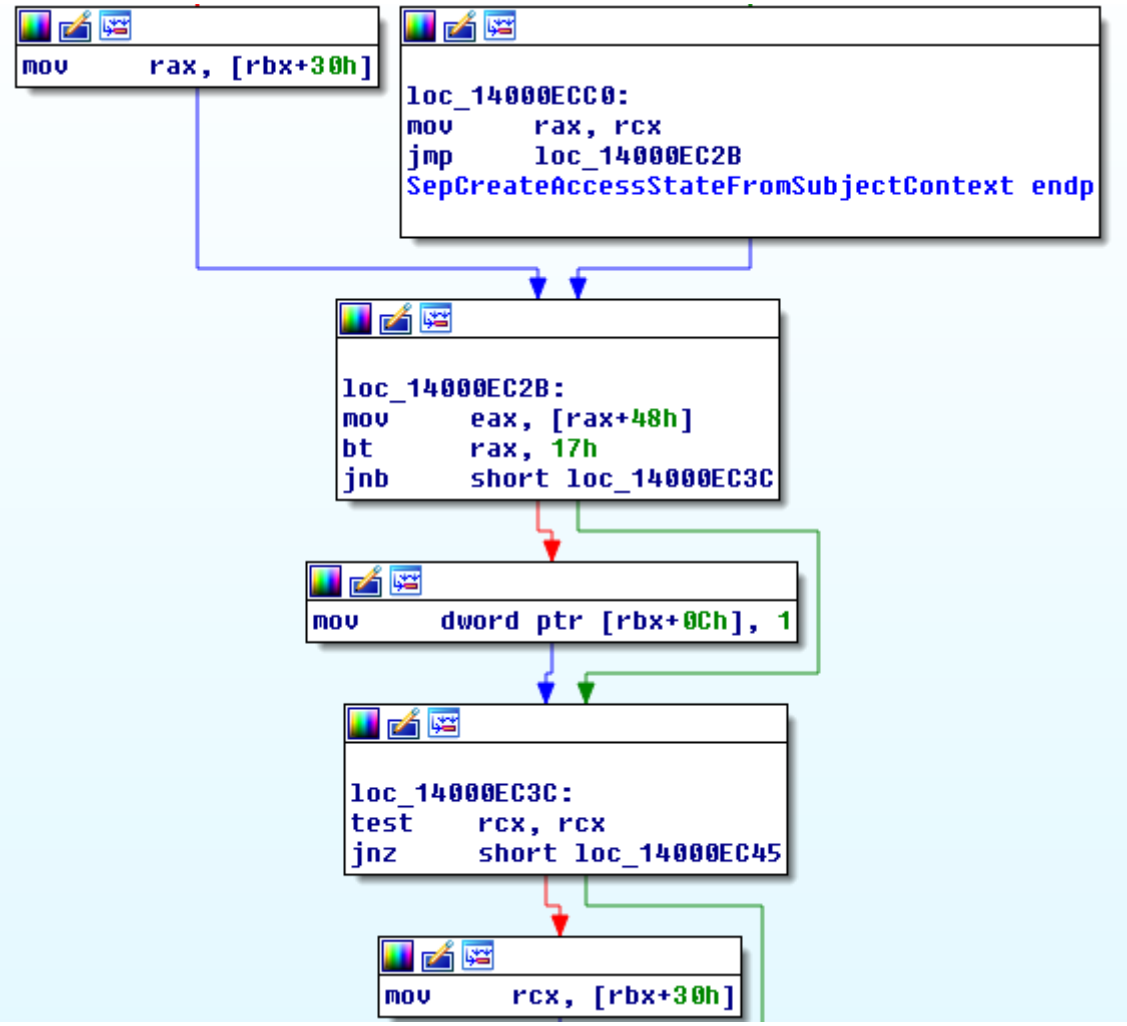
In this case we can just enable privileges at will by writing to *_SEP_TOKEN_PRIVILEGES.Enabled* member which will give us full access to an elevated

process. Again, injecting a remote thread will do the job. This will work fine up to **Windows 10 v1607**.

Recently, though, I noticed that my exploit stopped working in the latest **NT kernel version 10.0.15063**. Since this technique is a pretty much solid one, it made me dig a bit deeper. I was too curious to see what changed, and too annoyed seeing my exploit fail once more.

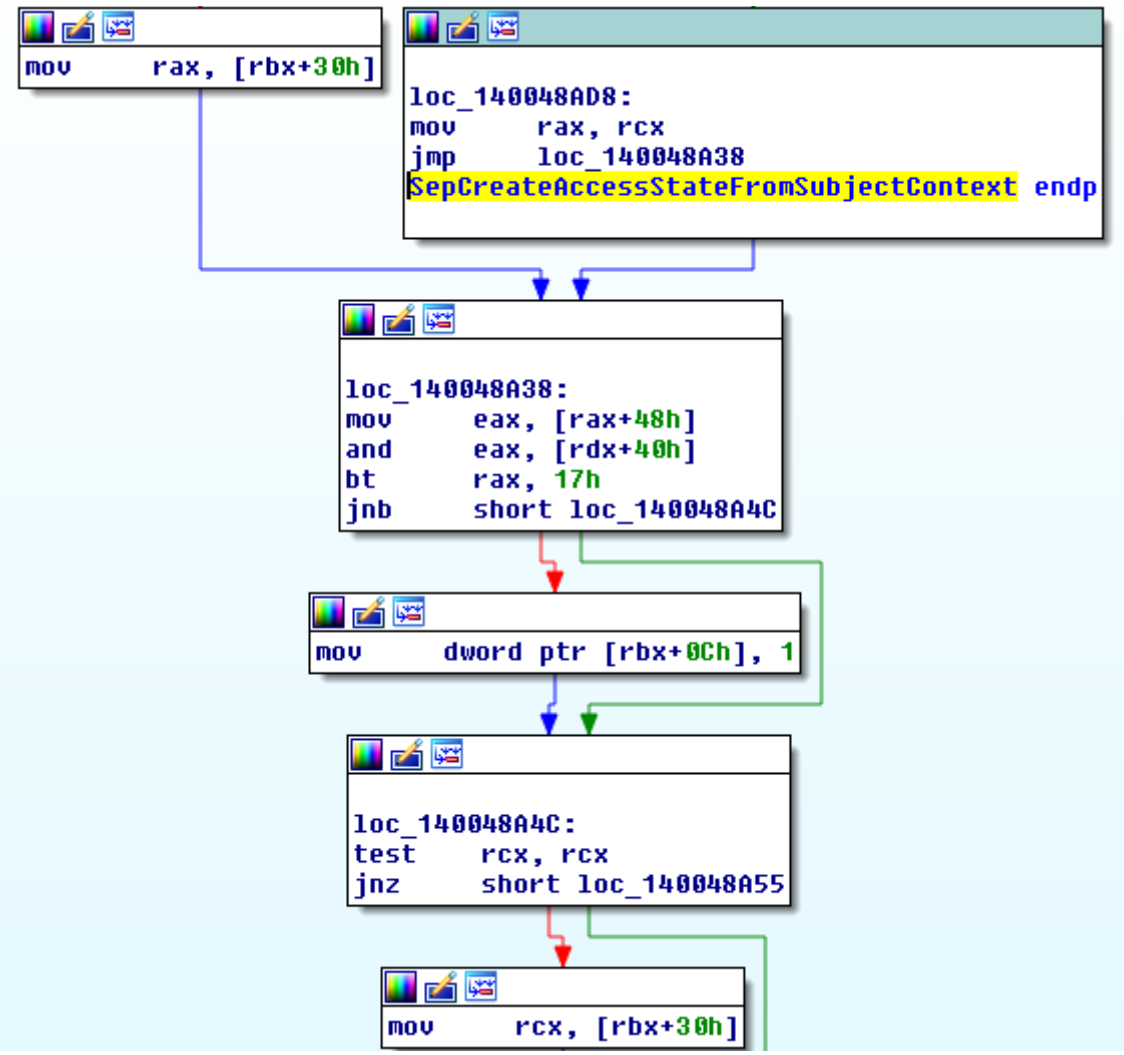
It seems that a tiny update in *nt!SepCreateAccessStateFromSubjectContext* function was enough to mess with my exploit. Indeed it seems that one added instruction makes a huge difference.

Let's first see the interesting part from version 10.0.14393:



At loc_14000EC2B we see that *_SEP_TOKEN_PRIVILEGES.Enabled* privileges value is being read in EAX. This will be taken into account to check if our process has enough privileges to interact with an elevated process, hence inject a remote thread.

Let's first see the interesting part from version 10.0.15063:



At loc_1400048A38 we see that *_SEP_TOKEN_PRIVILEGES.Enabled* privileges value is being read in EAX. However, this time that value will be ANDed with *_SEP_TOKEN_PRIVILEGES.Present*.

That member keeps the privileges that are present (not enabled) in the token of our process. This value basically indicates what privileges are available for our process to be enabled or later disabled of course.

Now, since our process is running as a limited user all of the interesting privileges are absent, so no matter what value you have set in the *_SEP_TOKEN_PRIVILEGES.Enabled* member, anything above the maximum available privileges for our process will be lost during this check since the result of that AND operation will be taken now in consideration and not the *SEP_TOKEN_PRIVILEGES.Enabled* value.

In order to make our exploit work again, we have to make sure that the privileges in *_SEP_TOKEN_PRIVILEGES.Present* are at least at the same level as those that we write in *SEP_TOKEN_PRIVILEGES.Enabled*, and then we win.

To sum up, I wouln't call this as a proper mitigation as basically it's not really blocking this exploitation method, but I can definitely imagine some people being annoyed by it, while looking at their kernel exploit suddenly fail.

And now you know!