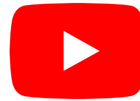
CVE-2019-12750: Symantec Endpoint Protection Local Privilege Escalation – Part 1

By Kyriakos Economou | December 3, 2019

A malicious application can take advantage of a vulnerability in Symantec Endpoint Protection to leak privileged information and/or execute code with higher privileges, thus taking full control over the affected host.

CVE-2019-12750: Symantec Endpoint Protection Local



Products Affected

- Symantec Endpoint Protection v14.x < v14.2 (RU1)
- Symantec Endpoint Protection v12.x < 12.1 (RU6 MP10)
- Symantec Endpoint Protection Small Business Edition v12.x < 12.1 (RU6 MP10c)

Introduction: Vulnerability Analysis – Methodology

A few months ago, while looking for a local privilege escalation vulnerability in the latest version of *Symantec Endpoint Protection* (*SEP v14.2 Build 2486*) software, we encountered a vulnerability that was hidden for several years.

In addition, the latest security updates around the kernel pool allocations that were introduced in Windows 10 v1809 gave us the opportunity to implement a different approach in order to successfully exploit this vulnerability in the latest version currently available; v1909.

Since the two approaches we used are quite different between them, we decided to split this write-up into two parts.

In the first part, we will be discussing the actual bug and how we took advantage of it in earlier Windows versions, Windows 7 to 10 v1803, without additional kernel mode execution control requirements.

In the second part, we will go through a more sophisticated approach that required further analysis of the vulnerable products due to the newly introduced *Low Fragmentation Heap (LFH)* for kernel mode pool allocations, in Windows 10 v1809 onwards, which broke the first exploitation method. This was necessary in order to obtain code execution in kernel mode while bypassing additional exploitation mitigations such as SMEP and KASLR.

Symantec Endpoint Protection – Vulnerability Analysis

When a new process is created, SEP injects a DLL module named *'sysfer.dll'* that makes a series of *input-output control (IOCTL)* requests once it's loaded. In this case we are interested at IOCTL 0x222014 that is sent to the *'SysPlant.sys'* kernel driver.

During the handling of this request, a programming mistake allows a malicious attacker to leak and corrupt kernel mode data. In particular, by examining one of the subroutines that are executed during this IOCTL request, we see that the following steps take place:

1. Call *ExAllocatePool* to allocate a buffer of 0x14 bytes in size in *Paged Pool*.
2. Call *IoAllocateMdl* to allocate a memory descriptor list for that buffer.
3. Call *MmProbeAndLockPages* to probe and lock the associated pages in memory.
4. Call *MmMapLockedPagesSpecifyCache* to map those pages into another virtual address range *(Figure 1)*.

```
loc_FFFFF806E13A6A15:                   ; DATA XREF: .rdata:FFFFF806E13C7F2C↓o
                mov     r8d, 1          ; CacheType
                mov     rcx, rsi        ; MemoryDescriptorList
                movzx   edx, r8b        ; AccessMode
                test    r14d, r14d
                jz      short loc_FFFFF806E13A6A68
                mov     r14d, edi
                mov     [rsp+58h+var_28], edi
                mov     eax, cs:dword_FFFFF806E13CD840
                or      eax, 10h
                mov     [rsp+58h+Priority], eax ; Priority
                mov     [rsp+58h+BugCheckOnFailure], edi ; BugCheckOnFailure
                xor     r9d, r9d        ; BaseAddress
                call    cs:MmMapLockedPagesSpecifyCache
```

Figure 1 – Mapping Paged Pool Chunk in User Mode

The last one of the steps described above maps that pool chunk in user mode in the context of the calling process. However, when a buffer is mapped the entire memory page where the buffer range applies is also mapped.

In this case, the buffer allocates a very small chunk that occupies in total 0x30 bytes (buffer size + pool header + alignment padding). This means that by mapping this pool chunk in user mode, we also leak the rest of the memory page. That is basically 0xFD0 (4048) bytes of additional kernel memory that may contain other privileged information.

Unfortunately that is not all. By default the memory page is mapped as writable, which also enables a user mode process to modify its contents. Any modification applied to the userland mapped memory page, will be reflected to original kernel mode memory page.

In order to mitigate the information leakage, the driver should allocate a kernel mode buffer that is of size multiple to the size of the memory page used by the system – usually 4KBs. This buffer must be sanitized (filled with zeros) before any mapping occurs.

As a side note, the kernel mode buffer should not be freed while it’s still mapped in user address space. Otherwise, any new data written to that buffer will also be leaked in userland.

As for the fact that the mapping of the kernel mode buffer should also be read-only, from Windows 8 onwards, the *MdlMappingNoWrite* flag can be used when calling the *MmMapLockedPagesSpecifyCache* function.

The first parameter passed to *MmMapLockedPagesSpecifyCache* is a pointer to an *MDL* structure that was allocated in the second step, as described above.

```
kd> dt nt!_MDL @rcx
+0x000 Next             : (null)
+0x008 Size             : 0n56
+0x00a MdlFlags         : 0n138
+0x00c AllocationProcessorNumber : 1
+0x00e Reserved         : 0
+0x010 Process          : (null)
+0x018 MappedSystemVa   : 0xffffb981`eebc1008 Void
+0x020 StartVa          : 0xffff8180`30573000 Void
+0x028 ByteCount        : 0x14
+0x02c ByteOffset       : 0x6b0
```

Figure 2 – MDL Structure

We are interested in the three highlighted members of this structure:

- ***StartVa*** – The starting address of the memory page
- ***ByteCount*** – Size of the buffer described by the MDL
- ***ByteOffset*** – Starting Offset of the buffer inside the memory page

The intention of the developer was to map that small buffer of 0x14 bytes of size into the user mode address space of the calling process. However, the system maps the entire memory page and not just the desired pool chunk, as showing below *(Figures 3, 4)*.

Figure 3 – Kernel Mode Buffer

Figure 4 – User Mode Mapped Buffer

This was a good finding in the first place, but we wanted to do something more with it, rather than just disclosing paged pool memory in userland.

Note that this works only once per process during the modules loading stage. So, we can't re-use the same IOCTL multiple times from the same process to map additional kernel memory pages in userland.

On the other side, since kernel objects are being allocated and freed all the time, the contents of the mapped page associated to those will be constantly changing. However, by re-using the

same IOCTL from our own function, once our process has been initialized, we can find the userland address where that chunk is mapped.

By knowing that, we can also find the base address in userland of that specific page by masking the last two bytes (VA & 0xFFFFF000), assuming page allocation granularity is 4KBs.

Once we have that information, we can parse the mapped page to find information and interesting objects that we can use to exploit this further.

Exploitation (Win 7 – Win 10 v1803)

In the first part of the exploitation write-up we will focus on Windows versions prior to Win 10 v1809. This will also help us appreciate during the second part, how *LFH* applied for kernel pool allocations affected our exploit in the latest Win 10 versions. So for this part, we will focus on Win 10 v1803.

Since we can only leak only one paged pool memory page per process, and different processes might as well leak the same page (if there are free chunks were more than one of 0x30 bytes chucks can fit), we started by launching a bunch of processes.

Each process uses the aforementioned IOCTL to get the base address of the mapped page in userland and save that data in a file for further investigation.

After creating a few processes, 'magic' started to happen.

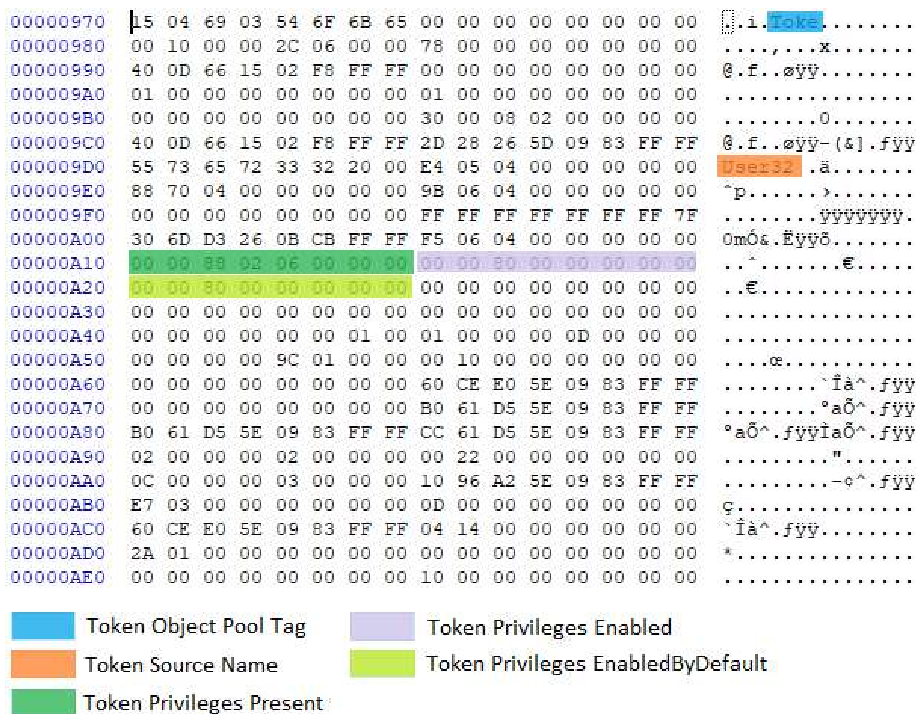


Figure 5 – Leaked Token Object

We noticed that occasionally, we were able to leak *Token Objects*, a well-known target for kernel LPE exploits. Yes, these are paged-pool objects, and now we own them.

Note that, the leaked memory pages do not necessarily contain object allocations associated with the calling process. In fact, usually there won't be any.

These are mainly random pages that are leaked when the aforementioned IOCTL request takes place, and the 0x30 bytes chunk is fitted wherever there is available space in paged kernel pool. That being said, the leaked *Token Object* does not necessarily belong to the calling process, and in fact usually it will not.

However, since we know that this interesting object type can be occasionally find itself inside a leaked page pool memory page, we need to find a way to take advantage of this situation.

Token Object Playground – First Method

Since every process has its own *Token Object*, we can start creating many processes which every time would be put in a

‘wait’ state. We need this to ensure that the associated token objects do not get freed.

In a few words, every child process will examine the leaked kernel memory page that is mapped in its address space and look for a leaked *Token Object*. Whatever the result is, the process will just wait afterwards indefinitely.

The more processes being created, the more *Token Objects* we will have allocated, and so eventually we will start leaking those in a writable usermode-mapped kernel page.

At this point we don’t know to which process each object belongs, and so we must keep a list of the process handles. Once the child process creation has finished, we can examine their tokens to find one where we have modified its original privileges and execute code in its own context.

This method works, but it can get better which brings us to the next one.

Token Object Playground – Second Method

Since our first objective is to leak as many *Token Objects* as possible, it makes sense that the more of these are allocated, the greater the chances are that an allocated 0x30 bytes chunk will find itself in the same memory page.

We can achieve this by using the *DuplicateTokenEx* function to create clones of the primary token of our exploit process. This function also allows us to specify that each clone will be a primary token which we can later use through the *CreateProcessAsUser* function. Alternatively, we can use the ImpersonateLoggedOnUser function to allow the calling thread of our exploit to use the elevated token privileges directly.

Once we have created a few thousand clones of our primary token, we can then fall back to the first method described above.

Again, we start creating child processes where each of these will examine the leaked kernel page mapped in its own address space and search for *Token Objects* to modify (see Figure 5).

Finally, we can go through our token clones list and use GetTokenInformation with the *TokenPrivileges* *TOKEN_INFORMATION_CLASS* and search for an instance that has privileges that only an elevated process can have, such as the "SeDebugPrivilege". Last but not least, we use the modified token with *CreateProcessAsUser* function to start a new process with elevated privileges, or through the *ImpersonateLoggedOnUser* function to elevate the privileges of the calling thread.

Summarizing the steps

1. Create a few thousand of *Primary Token* clones
2. Start self-examining child processes that search and modify leaked *Token Objects*
3. Parse our *Primary Token* clones list to find one that was modified
4. Use *CreateProcessAsUser* or *ImpersonateLoggedOnUser* functions to execute code with higher privileges
5. Inject and execute code in a process running as SYSTEM

Conclusion

This finding is an example of how a simple programming mistake can turn into a serious vulnerability. The developer needed to map in the calling process a small chunk of data without considering all the details of the documentation and their possible security implications.

In the second part of exploiting this vulnerability, we will focus on the latest Windows 10 v1909 and see how kernel pool LFH addition forced to us to look for other ways for exploiting this. Stay tuned.

Timeline

1. Date of discovery: April 2019
2. Vendor informed: 18 April 2019
3. Vendor Acknowledged: 19 April 2019
4. Vendor Requested Extra Time: 19 April 2019
5. Advisory: 31 July 2019
6. Nettitude blog: 5 December 2019