13 October 2016

Analysing the NULL SecurityDescriptor Kernel Exploitation Mitigation in the latest Windows 10 v1607 Build 14393

Written By

Kyriakos (kyREcon) Economou

For

Nettitude Labs

tl;dr: We recently discovered a new and quietly released Windows kernel exploitation defence. Exploiting a kernel bug by setting the pointer to the SecurityDescriptor to NULL in the header of a process object running as SYSTEM won't work from Windows 10 v1607 (Build 14393). If you want to know why, keep reading.

Introduction

One of the most efficient and reliable attack vectors used by kernel privilege escalation exploits, is to set the pointer to the *SecurityDescriptor* of a securable object to NULL. This pointer resides in the header of the object.

By performing this action against a process that runs with high privileges such as the SYSTEM account, the exploit process can then inject a remote thread and execute code under the security context of the target process.

This kernel object attack vector is quite powerful also for the reason that makes *Supervisor Mode Execution Protection* (SMEP) and other protections such as Non-Executable Kernel pool memory totally obsolete, since we don't have to execute our payload anymore in Kernel mode and/or address space in order to elevate our privileges.

In other words, zeroing out the pointer to the security descriptor of a process running as SYSTEM is pretty much equal to a 'GOD_MODE' cheat in the language of video game players.

In Windows 10 using this attack vector with just a write-zero-primitive was only possible until v1511 (Build 10586). Indeed, in the latest Windows 10 v1607 (Build 14393), a mitigation against this attack vector has been added. This doesn't apply in earlier major Windows versions such as Windows 8.1 and below.

Note: This analysis is based on Windows 10 v1607 (Build 14393) x64.

The SecurityDescriptor

Every kernel object is associated with a header structure that is prepended to the object itself. One of the members of this structure is a pointer to a *SecurityDescriptor* which contains the *Access Control List* (ACL) for that object. The *ACL* contains the information about who can access this very object and with what permissions.

As a side note, not all objects store the pointer to their SecurityDescriptor in their header. For example persistent objects such as files and registry keys use their own mechanism to access this data during the access check to the object.

Using Windbg, let's get some information about the process of Windows Explorer.

```
kd> !process 0 0 explorer.exe

PROCESS ffffe18884ac4800 ← Address of the process object.
SessionId: 1  Cid: 0c58    Peb: 00dc7000  ParentCid: 0c2c
DirBase: 00ce8000 ObjectTable: ffffa60587c89400  HandleCount: <Data Not Accessible>
Image: explorer.exe
```

Figure 1. Getting address of process object

Now let's get some information about its header.

```
kd> !object ffffe18884ac4800

Object: ffffe18884ac4800  Type: (ffffe1887e4b5b60) Process
 ObjectHeader: ffffe18884ac47d0 ← object header address
HandleCount: 10  PointerCount: 359279
```

Figure 2. Getting address of object header

Let's now get some more information about this structure.

```
kd> dt nt!_OBJECT_HEADER ffffe18884ac47d0

   +0x000 PointerCount    : 0n359279
   +0x008 HandleCount     : 0n10
+0x008 NextToFree      : 0x00000000`0000000a Void
+0x010 Lock            : _EX_PUSH_LOCK
+0x018 TypeIndex       : 0x18 ''
+0x019 TraceFlags      : 0 ''
+0x019 DbgRefTrace     : 0y0
+0x019 DbgTracePermanent : 0y0
+0x01a InfoMask        : 0x88 ''
+0x01b Flags           : 0 ''
+0x01b NewObject       : 0y0
+0x01b KernelObject    : 0y0
+0x01b KernelOnlyAccess : 0y0
+0x01b ExclusiveObject  : 0y0
+0x01b PermanentObject  : 0y0
+0x01b DefaultSecurityQuota : 0y0
+0x01b SingleHandleEntry : 0y0
+0x01b DeletedInline    : 0y0
+0x01c Reserved        : 0xfc08548
+0x020 ObjectCreateInfo : 0xffffe188`84389100 _OBJECT_CREATE_INFORMATION
+0x020 QuotaBlockCharged : 0xffffe188`84389100 Void
+0x028 SecurityDescriptor : 0xffffa605`875dac8e Void ← pseudo pointer to the object Security Descriptor
+0x030 Body            : _QUAD ← start of Process Object (ffffe18884ac4800)
```

Figure 3. Getting SecurityDescriptor pointer

As you can see by knowing the address of our object, we can easily get the address where the pointer to the object security descriptor is stored. This is basically located at address: *OBJECT_ADDRESS – sizeof(ULONG_PTR)*.

Going back to the pointer to that security descriptor, remember this address is practically a pseudopointer, meaning that its last 4 bits contain information that is not related with the actual address. To get the actual address, we need to mask those bits. This can be achieved by doing *(PSEUDO_POINTER & 0xFFFFFFFFFFFFFFF0)*.

So let's see what information is stored there in this case.

```
1: kd> !sd (0xffffa605875dac8e & 0xFFFFFFFFFFFFFFF0)
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8814
            SE_DACL_PRESENT
            SE_SACL_PRESENT
            SE_SACL_AUTO_INHERITED
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-325928935-2433392120-1840313195-1000
->Group   : S-1-5-21-325928935-2433392120-1840313195-513
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x5c
->Dacl    : ->AceCount   : 0x3
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001fffff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-325928935-2433392120-1840313195-1000

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001fffff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18

->Dacl    : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[2]: ->AceFlags: 0x0
->Dacl    : ->Ace[2]: ->AceSize: 0x1c
->Dacl    : ->Ace[2]: ->Mask : 0x00121411
->Dacl    : ->Ace[2]: ->SID: S-1-5-5-0-128997

->Sacl    :
->Sacl    : ->AclRevision: 0x2
->Sacl    : ->Sbz1       : 0x0
->Sacl    : ->AclSize    : 0x1c
->Sacl    : ->AceCount   : 0x1
->Sacl    : ->Sbz2       : 0x0
->Sacl    : ->Ace[0]: ->AceType: SYSTEM_MANDATORY_LABEL_ACE_TYPE
->Sacl    : ->Ace[0]: ->AceFlags: 0x0
->Sacl    : ->Ace[0]: ->AceSize: 0x14
->Sacl    : ->Ace[0]: ->Mask : 0x00000003
->Sacl    : ->Ace[0]: ->SID: S-1-16-8192
```

Figure 4. Reading the Security Descriptor

Let's see now how this translates from a high level perspective.

High level overview

By using [ProcessExplorer] we can get an idea of who has access to the process object and at what level. So let's see how the ACL looks in ProcessExplorer.

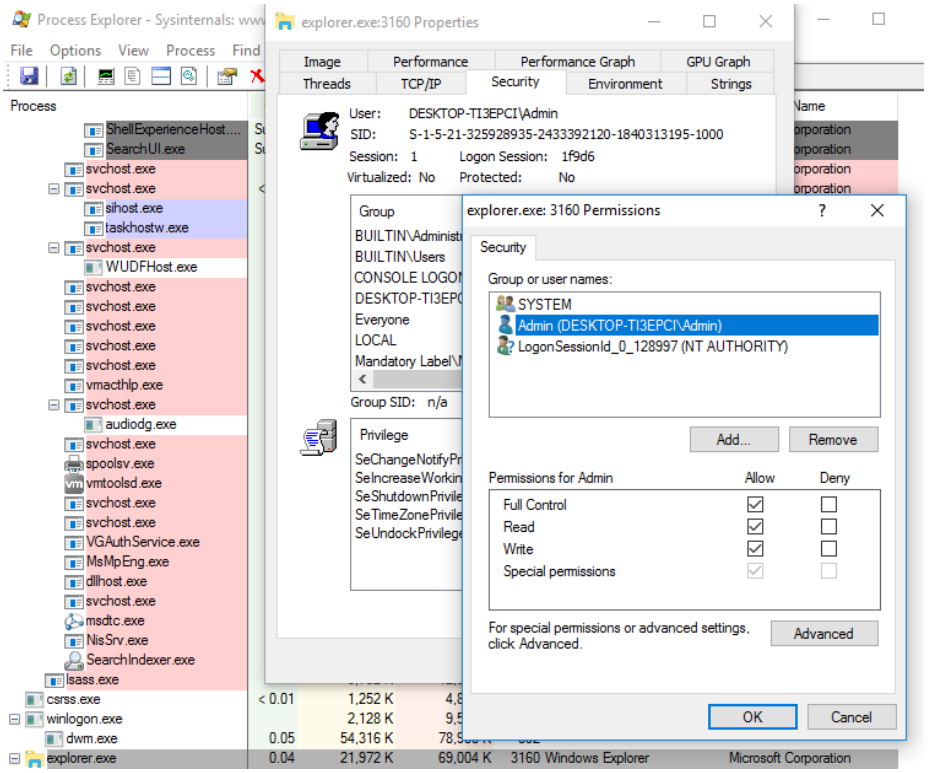

Figure 5. Windows Explorer Default Permissions

Indeed, this matches the information that we retrieved through the security descriptor. The SYSTEM and the current user (Admin) accounts have full access permissions (0x1FFFFF) to this process object.

NULL SecurityDescriptor Attack Vector

So up to Windows 10 v1511 this is what you would see after using the aforementioned attack vector:

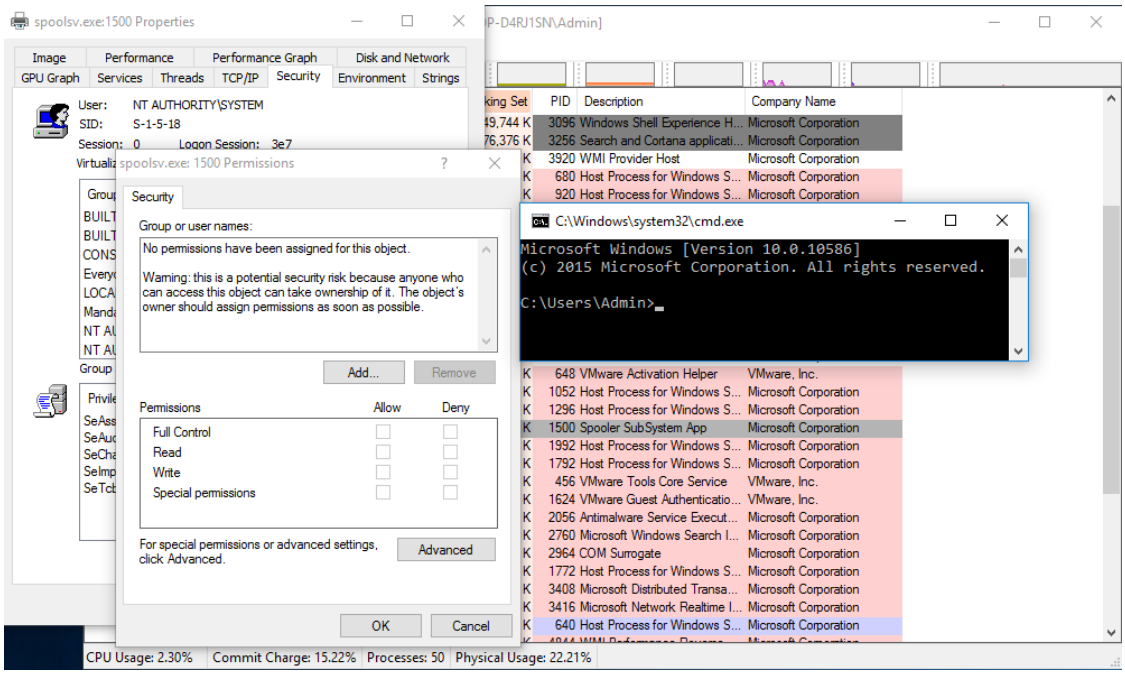

Figure 6. Win10 v1511 – NULL SecurityDesscriptor Pointer

The target process spoolsv.exe is running as SYSTEM, but there is no information about who can access this object and at what level. This grants full access to everyone.

Remember, this is not the same as pointing to a valid security descriptor with no *Access Control Entries* (ACEs) defined, thus an empty *Discretionary Access Control List* (DACL) . In that case access would be denied to everyone.

The figure above, depicts the result of a kernel exploit using a write-zero-primitive to set the *SecurityDescriptor* pointer member of the header of the target process object to zero. In this case when someone requests access to the process object, if that member is set to zero the kernel assumes that this object was created with a NULL *DACL*, and will grant access to everyone.

Mitigating the NULL SecurityDescriptor Attack Vector

In the latest Windows 10 v1607, this attack vector is now blocked. This is achieved by performing a simple check against a global table describing some characteristics of the NT

kernel object types whenever a process attempts to get a handle, thus get access to an object.

So this is how this attack type was mitigated in a matter of a few screenshots.

```
loc_FFFFF800E2BFD5AF:
xor     r13d, r13d
mov     rax, rdi
shr     rax, 8
movzx   ecx, al
movzx   eax, byte ptr [rdi+18h]
xor     rcx, rax
mov     [rbp+120h+var_144], r13d
xor     rcx, rdx
mov     [rbp+120h+var_E0], r13
mov     rax, rdi
shr     rax, 8
mov     r12, ds:rva ObTypeIndexTable[r8+rcx*8]
movzx   ecx, al
movzx   eax, byte ptr [rdi+18h]
xor     rcx, rax
mov     [rbp+120h+var_150], r12
xor     rcx, rdx
lea     rax, SeDefaultObjectMethod ; load to RAX address of nt!SeDefaultObjectMethod function
mov     r14, ds:rva ObTypeIndexTable[r8+rcx*8] ; ;read address of ObjectType object
mov     [rbp+120h+var_168], r14
cmp     [r14+98h], rax  ; Check that these two addresses are the same.
jnz     loc_FFFFF800E2BFDA3F
```

Figure 7. Checking which if the routine used to manage the security of the object is the default

```
prefetchw byte ptr [rdi+28h]
mov     rdx, [rdi+28h]  ; read on RDX ObjectHeader.SecurityDescriptor
test    dl, 0Fh         ; check that not all last 4 bits are zero
jz      short loc_FFFFF800E2BFD62E ; jump if they are. Will always be the case if our exploit sets the SecurityDescriptor to zero.
```

Figure 8. Checking if last 4 bits of the pointer to the SecurityDescriptor are set to zero

```
loc_FFFFF800E2BFD62E:   ; save SecurityDescriptor pointer to RBX
mov     rbx, rdx
and     edx, 0Fh        ; zero low-dword of RDX, except from the 4 last bits. These will be also zero in case of the described attack.
and     rbx, 0FFFFFFFFFFFFFFF0h ; zero out last 4 bits of RDX. Remember these are the bits that are irrelevant with the actual pointer to the SecurityDescriptor.
cmp     edx, r10d       ; when we arrive here r10d == 1
jb      loc_FFFFF800E2BFDC6F ; jump if edx < 1, which will be the case if the SecurityDescriptor was set to NULL.
```

Figure 9. Checking if the low DWORD of the pointer to the SecurityDescriptor is less than one

```
loc_FFFFF800E2BFDC6F:   ; check if RBX!=0. It won't be the case
test    rbx, rbx
jnz     loc_FFFFF800E2BFDD20
```

Figure 10. Checking if the full pointer to the SecurityDescriptor is not zero

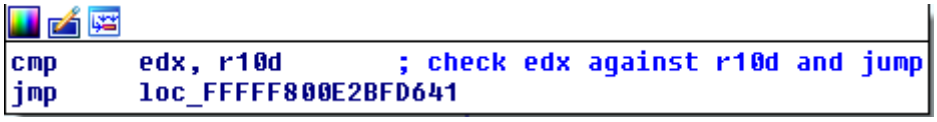

Figure 11. Another check of the low DWORD of the pointer with r10d register

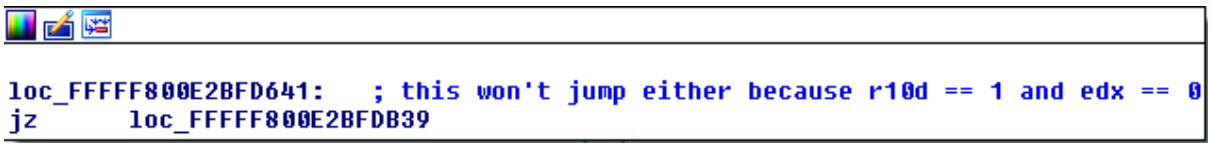

Figure 12. Based on the check in the previous figure

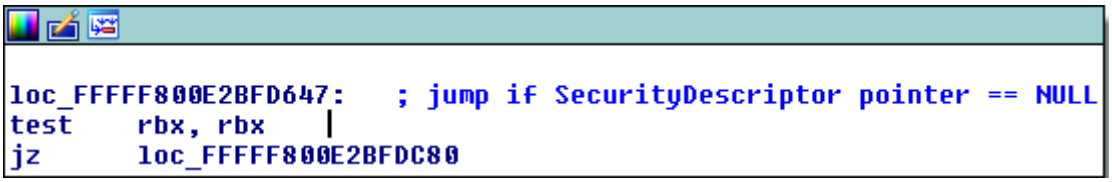

Figure 13. Checking if the entire pointer to the SecurityDescriptor is zero

The next check is the crucial one, so we will explain some things first.

If you have a look at Figure 7, you will notice that *R14* register holds the address of an *ObjectType* structure. The Kernel maintains a table of pointers *(ObTypeIndexTable)* to these structures to keep information about the various object types.

So let's examine the information that this *ObjectType* structure contains.

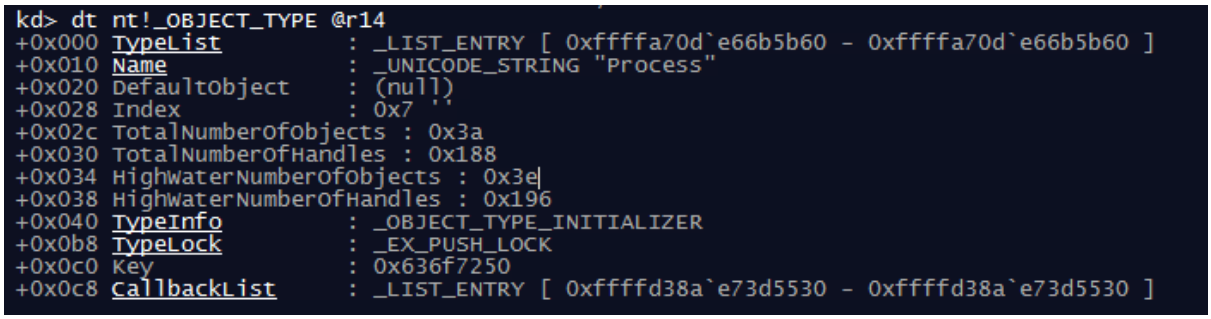

Figure 14. ObjectType Object

Notice the index value is set to 7. This indicates the index where a pointer to this *ObjectType* structure is stored inside the *ObTypeIndexTable*, and we also know that the object we are trying to access is a process object.

Let's move on now with the next check as show below.

```
PAGE:FFFFF800E2BFDC80 loc_FFFFF800E2BFDC80:                   ; CODE XREF: ObpCreateHandle+79A↑j
PAGE:FFFFF800E2BFDC80                 test    byte ptr [r14+42h], 8
PAGE:FFFFF800E2BFDC85                 jnz     loc_FFFFF800E2D90937
PAGE:FFFFF800E2BFDC8B                 test    byte ptr [rdi+1Ah], 2
PAGE:FFFFF800E2BFDC8F                 jz      loc_FFFFF800E2BFD650
PAGE:FFFFF800E2BFDC95                 jmp     loc_FFFFF800E2D90937
```

Figure 15. Checking if ObjectType.SecurityRequired flag is set

We see that it checks if the 3rd bit at [r14+0x42] is not zero and in that case it will jump.

So let's examine the *TypeInfo* member of the *ObjectType* structure, since that offset is part of it.

```
1: kd> dx -r1 (*((ntkrnlmp!_OBJECT_TYPE_INITIALIZER *)0xffffa70de66b5ba0))
(*((ntkrnlmp!_OBJECT_TYPE_INITIALIZER *)0xffffa70de66b5ba0))                 [Type: _OBJECT_TYPE_INITIALIZER]
    [+0x000] Length           : 0x78 [Type: unsigned short]
    [+0x002] ObjectTypeFlags  : 0xca [Type: unsigned char]
    [+0x002 ( 0: 0)] CaseInsensitive  : 0x0 [Type: unsigned char]
    [+0x002 ( 1: 1)] UnnamedObjectsOnly : 0x1 [Type: unsigned char]
    [+0x002 ( 2: 2)] UseDefaultObject : 0x0 [Type: unsigned char]
    [+0x002 ( 3: 3)] SecurityRequired : 0x1 [Type: unsigned char]
    [+0x002 ( 4: 4)] MaintainHandleCount : 0x0 [Type: unsigned char]
    [+0x002 ( 5: 5)] MaintainTypeList : 0x0 [Type: unsigned char]
    [+0x002 ( 6: 6)] SupportsObjectCallbacks : 0x1 [Type: unsigned char]
    [+0x002 ( 7: 7)] CacheAligned     : 0x1 [Type: unsigned char]
```

Figure 16. ObjectType.TypeInfo

So basically, if the *SecurityRequired* flag is set, which it is, then the first jnz jump will be taken and the kernel will trigger a BSoD with a *"BAD OBJECT HEADER"* stop code.

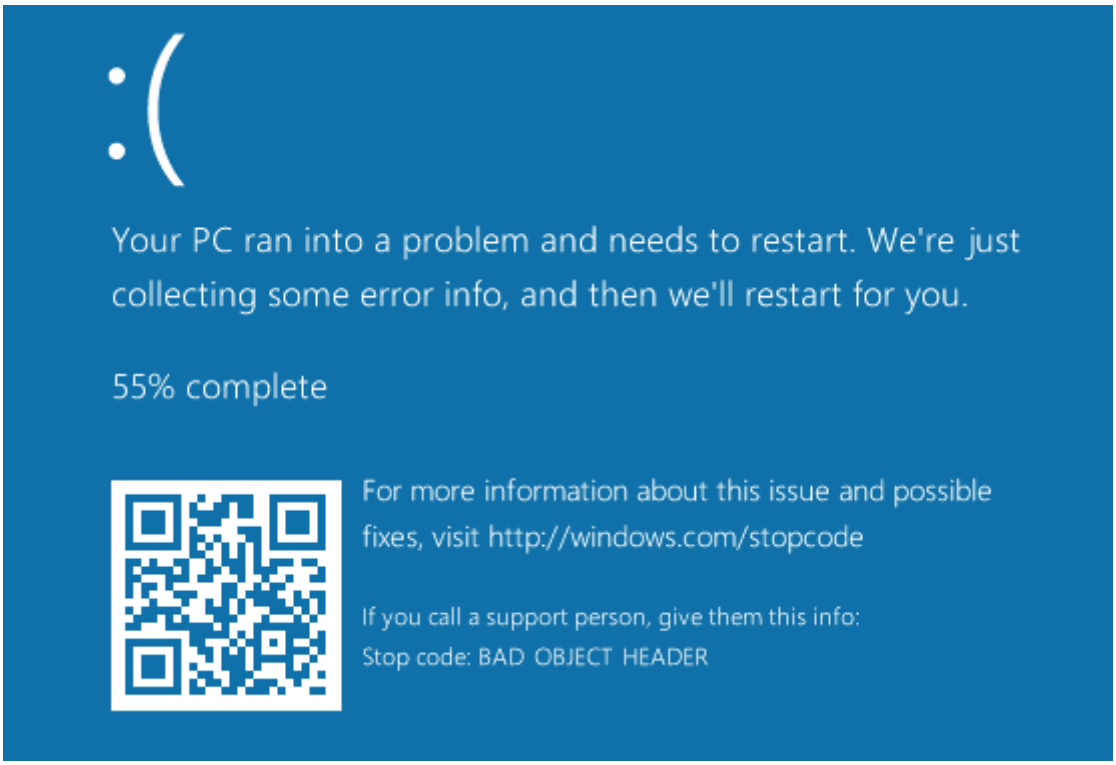

Figure 17. BSoD – BAD OBJECT HEADER

Here is the code that triggers the crash:

```
loc_FFFFF800E2D90937:
xor     eax, eax
mov     r8, r14         ; BugCheckParameter2
mov     rdx, rdi        ; BugCheckParameter1
mov     qword ptr [rsp+218h+PreviouslyGrantedAccess], rax ; BugCheckParameter4
mov     ecx, 189h       ; BugCheckCode
lea     r9d, [rax+1]    ; BugCheckParameter3
call    KeBugCheckEx
; END OF FUNCTION CHUNK FOR ObpCreateHandle
```

Figure 18. Triggering BSoD

Putting everything together, the kernel will now perform an extra security check before even examining if the process that requests access to an object it actually can access the object based on the *SecurityDescriptor* of the object and the security token of the requestor.

This could be summed up with the following:

```
if(ObjectHeader.SecurityDescriptor == NULL && (ObjectType.SecurityRequired ||
(ObjectHeader.InfoMask &2) != 0))
{
    BugCheckEx(BAD_OBJECT_HEADER);
}
```

Objects created with NULL DACL

In case due to a programming error, bad security practice, or for any other reason an object is created with a NULL *DACL* (which grants access to everyone), this won't have any impact in the host.
The reason for this, is that even if we decide to assign such *DACL* to the security descriptor, the *SecurityDescriptor* pointer in the header of the object will still be valid and it will point to a *SecurityDescriptor* with a *DACL* set to NULL.

An example is shown below.

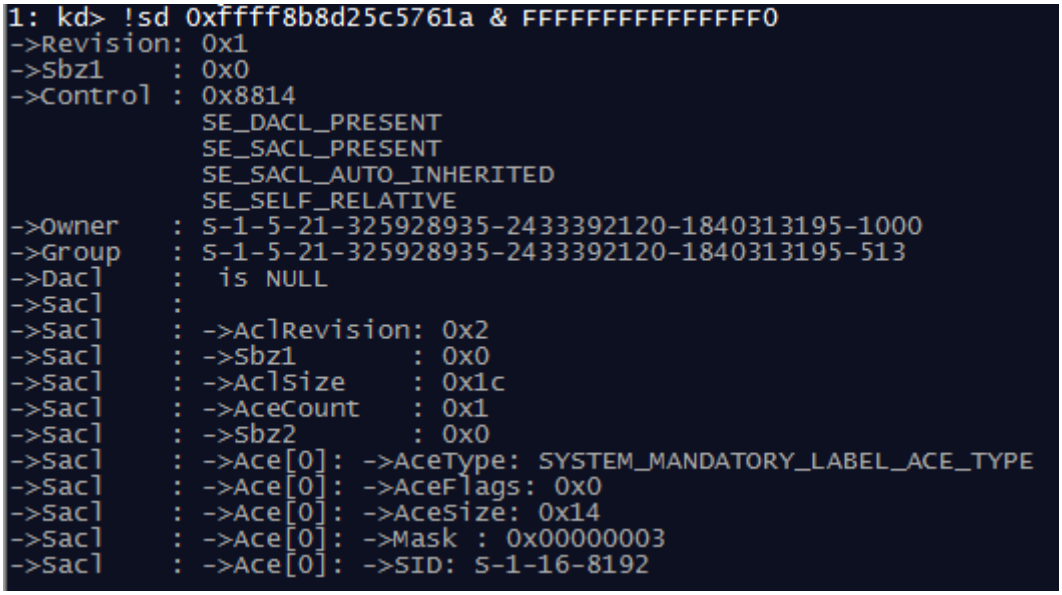

Figure19. Object created with a NULL DACL

Final thoughts

It is clear that MS guys are continuously trying to mitigate various attack vectors. The examined attack vector that has been now blocked, is one of the most used ones when it comes to kernel bugs exploitation.

However, as already mentioned this security check is only effective when the bug only allows to write a zero in an arbitrary kernel memory address, and the chosen attack vector is the discussed one.

If the bug allows for a read-write primitive as well, then we could leak the pointer to the security descriptor and set a NULL DACL there which would pass under the radar.

On the other hand, if we can control the value that we are writing to an arbitrary kernel memory address then we could choose different exploitation paths.

One of them, would be to access the token of the exploit process and enable privileges at will, which pretty much would have the same “GOD_MODE” effect.