EXPLOITING WDM AUDIO DRIVERS

Reversemode
Advanced Reverse Engineering Services

Rubén Santamarta
Reversemode
December 2007

1.0

..
In memory of Carlos P.
You'll never walk alone.
..

1. Abstract

This paper covers an attack vector which is inherent to certain WDM audio drivers running on Windows Vista, XP, 2000 and 2003.

It is oriented towards researchers and developers with the aim of helping them to keep their code safe or identify vulnerabilities. The author does not assume any responsibility for the illegal usage of the information provided.

2. Introduction

Nowadays, writing drivers for sound cards is even not necessary in certain cases, whenever the manufactured device is fully compliant with the industry standards already implemented by Microsoft Windows. Anyway, if a driver is still necessary Microsoft provides the WDM that reduces the complexity of the task. Specifically, for audio and video drivers, Windows includes the Kernel Streaming Architecture, which allows developers to write drivers in an easy way since most of the logic is implemented within Windows core components.

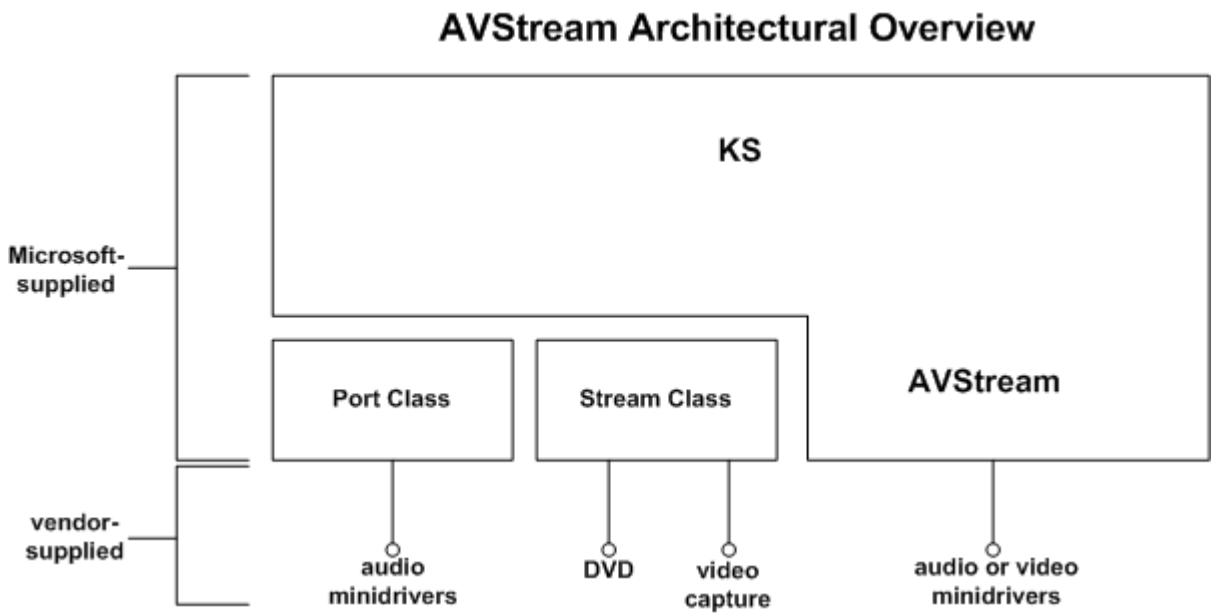

Image 1. Kernel Streaming architecture (msdn).

As the above image shows, developers facing a WDM audio driver should choose between drivers binded to the Port Class driver(portcls.sys) or directly to the KS (ks.sys) . In many cases, PortCls.sys is the preferred one.

So, let's see what would be the common initialization sequence for these type of drivers:

1. Bind to the desired driver, I.e PortCls.sys , at Driver's entry point.

```
extern  "C" NTSTATUS DriverEntry
(
        PDRIVER_OBJECT DriverObject,
        PUNICODE_STRING RegistryPath
)

{
        return PcInitializeAdapterDriver ( DriverObject,
                                           RegistryPath,
                                           XpAddDevice );
}
```

Reversing PcInitializeAdapterDriver we see the following:

module: portcls.sys

```
1.
2.  mov     ecx, [edi+18h]
3.  push    esi
4.  mov     esi, ds:__imp__KsSetMajorFunctionHandler@8 ;
    KsSetMajorFunctionHandler(x,x)
5.  mov     [ecx+4], eax
6.  push    0Eh             ; MajorFunction
7.  push    edi             ; DriverObject
8.  mov     dword ptr [edi+34h], offset ?KsoNullDriverUnload@@YGXPAU_DRIVER_OBJECT@@@Z
    ; KsoNullDriverUnload(_DRIVER_OBJECT *)
9.  mov     dword ptr [edi+0A4h], offset ?
    DispatchPnp@@YGJPAU_DEVICE_OBJECT@@PAU_IRP@@@Z ; DispatchPnp(_DEVICE_OBJECT *,_IRP
    *)
10. mov     dword ptr [edi+90h], offset ?
    DispatchPower@@YGJPAU_DEVICE_OBJECT@@PAU_IRP@@@Z ; DispatchPower(_DEVICE_OBJECT
    *,_IRP *)
11. mov     dword ptr [edi+94h], offset ?
    PerfWmiDispatch@@YGJPAU_DEVICE_OBJECT@@PAU_IRP@@@Z ;
    PerfWmiDispatch(_DEVICE_OBJECT *,_IRP *)
12. mov     dword ptr [edi+38h], offset ?
    DispatchCreate@@YGJPAU_DEVICE_OBJECT@@PAU_IRP@@@Z ; DispatchCreate(_DEVICE_OBJECT
    *,_IRP *)
13. call    esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunctionHandler(x,x)
14. push    3               ; MajorFunction
15. push    edi             ; DriverObject
16. call    esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunctionHandler(x,x)
17. push    4               ; MajorFunction
18. push    edi             ; DriverObject
19. call    esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunctionHandler(x,x)
20. push    9               ; MajorFunction
21. push    edi             ; DriverObject
22. call    esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunctionHandler(x,x)
23. push    2               ; MajorFunction
24. push    edi             ; DriverObject
25. call    esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunctionHandler(x,x)
26. push    14h             ; MajorFunction
27. push    edi             ; DriverObject
28. call    esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunctionHandler(x,x)
29. push    15h             ; MajorFunction
30. push    edi             ; DriverObject
31. call    esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunctionHandler(x,x)
32. xor     eax, eax
33. pop     esi
34. jmp     short loc_25F74
```

The function installs directly IRP handlers for :

- IRP_MJ_PNP (9)
- IRP_MJ_POWER (10)
- IRP_MJ_SYSTEM_CONTROL (11)
- IRP_MJ_CREATE (12)

Another seven remaining handlers are installed by using *KsSetMajorFunctionHandler*. It's interesting to see what the msdn says about this function:

*The **KsSetMajorFunctionHandler** function sets the handler for a specified major function to use the internal dispatching. It routes through a **KSDISPATCH_TABLE** contained in the opaque object header to be the first element within a structure pointed to by an **FsContext** within a file object. The dispatching assumes the table and **FsContext** structure are initialized by the device using **KsAllocateObjectHeader**.*

The verb "to assume" probably sounds really scary for all of you. Historically, software assuming things like fixed buffer sizes, inputs... has lead to important vulnerabilities.

How does the structure KSDISPATCH_TABLE look like?

```
Typedef struct{
  PDRIVER_DISPATCH DeviceIoControl;       //    IRP_MJ_DEVICE_CONTROL
  PDRIVER_DISPATCH Read;                  //    IRP_MJ_READ
  PDRIVER_DISPATCH Write;                 //    IRP_MJ_WRITE
  PDRIVER_DISPATCH Flush;                 //    IRP_MJ_FLUSH
  PDRIVER_DISPATCH Close;                 //    IRP_MJ_CLOSE
  PDRIVER_DISPATCH QuerySecurity;         //    IRP_MJ_QUERY_SECURITY
  PDRIVER_DISPATCH SetSecurity;           //    IRP_MJ_SET_SECURITY
  PFAST_IO_DEVICE_CONTROL FastDeviceIoControl; //
DriverObject.FastIoDispatch.FastIoDeviceControl
  PFAST_IO_READ FastRead;           //    DriverObject.FastIoDispatch.FastRead
  PFAST_IO_WRITE FastWrite;         //     DriverObject.FastIoDispatch.FastWrite
}KSDISPATCH_TABLE, *PKSDISPATCH_TABLE;
```

```
KsSetMajorFunctionHandler(
      IN PDRIVER_OBJECT DriverObject,
      IN ULONG MajorFunction
      );
```

KsSetMajorFunctionHandler is quite simple, it installs the proper IRP handler for *MajorFunction* >= 0, otherwise installing the *FastIo* handlers.

module ks.sys

```
[...]

PAGE:00026FA2
PAGE:00026FA2 loc_26FA2:      ; CODE XREF: KsSetMajorFunctionHandler(x,x)+6Bj
PAGE:00026FA2                 mov     ecx, offset _DispatchRead@8 ; DispatchRead(x,x)
PAGE:00026FA7                 jmp     short loc_26FE4
PAGE:00026FA9 ;
---------------------------------------------------------------------------
PAGE:00026FA9
PAGE:00026FA9 loc_26FA9:      ; CODE XREF: KsSetMajorFunctionHandler(x,x)+68j
PAGE:00026FA9                 mov     ecx, offset _DispatchClose@8 ; DispatchClose(x,x)
PAGE:00026FAE                 jmp     short loc_26FE4
PAGE:00026FB0 ;
---------------------------------------------------------------------------
PAGE:00026FB0
PAGE:00026FB0 loc_26FB0:      ; CODE XREF: KsSetMajorFunctionHandler(x,x)+64j
PAGE:00026FB0                 mov     ecx, offset _DispatchCreate@8 ;
DispatchCreate(x,x)
PAGE:00026FB5                 jmp     short loc_26FE4
PAGE:00026FB7 ;
---------------------------------------------------------------------------
PAGE:00026FB7
PAGE:00026FB7 loc_26FB7:      ; CODE XREF: KsSetMajorFunctionHandler(x,x)+5Dj
PAGE:00026FB7                 mov     ecx, offset _DispatchFlush@8 ; DispatchFlush(x,x)
PAGE:00026FBC                 jmp     short loc_26FE4

[...]

PAGE:00026FE4                 mov     edx, [ebp+DriverObject]
PAGE:00026FE7                 mov     [edx+eax*4+38h], ecx
                              ; DrvObj->MajorFunction[IRP_MJ_XXX] = DispatchXXXXX
[...]
PAGE:00026F74                 mov     eax, [ebp+DriverObject]
PAGE:00026F77                 mov     eax, [eax+28h] ; DriverObject->FastIoDispatch
PAGE:00026F7A                 mov     dword ptr [eax+8], offset _DispatchFastRead@32 ;
DispatchFastRead(x,x,x,x,x,x,x,x)
```

```
[...]
```

The interesting part is the content of the installed IRP handlers: DispatchRead,DispatchWrite...

Let's see an example:

```
module: ks.sys
PAGE:0002B578 _DispatchWrite@8 proc near               ; DATA XREF:
KsSetMajorFunctionHandler(x,x)+70o
PAGE:0002B578
PAGE:0002B578 PDEVICE_OBJECT  = dword ptr  8
PAGE:0002B578 PIRP            = dword ptr  0Ch
PAGE:0002B578
PAGE:0002B578                 mov     edi, edi
PAGE:0002B57A                 push    ebp
PAGE:0002B57B                 mov     ebp, esp
PAGE:0002B57D                 mov     ecx, [ebp+PIRP]
PAGE:0002B580                 mov     eax, [ecx+60h]; Irp->CurrentStackLocation
PAGE:0002B583                 mov     eax, [eax+18h]; IrpSp->FileObject
PAGE:0002B586                 mov     eax, [eax+0Ch]; FileObject->FsContext
PAGE:0002B589                 mov     eax, [eax]    ; *FsContext
PAGE:0002B58B                 mov     eax, [eax]    ; KSDISPATCH_TABLE
PAGE:0002B58D                 push    ecx
PAGE:0002B58E                 push    [ebp+PDEVICE_OBJECT]
PAGE:0002B591                 call    dword ptr [eax+8];KsDsp->Write(pDev,pIrp);
PAGE:0002B594                 pop     ebp
PAGE:0002B595                 retn    8
PAGE:0002B595 _DispatchWrite@8 endp
```

These handlers are merely a wrapper that uses the KSDISPATCH_TABLE for routing the IRP. Now, remember what the MSDN said:

> *The dispatching assumes the table and **FsContext** structure are initialized by the device using **KsAllocateObjectHeader**.*

What would happen if the device, by any reason, has not properly initialized *FsContext* ? **EAX == NULL**.

Therefore, by issuing a Write (Read, DeviceIoControl...) operation in the exposed device, the default driver's IRP handler DispatchWrite will dereference the pointer to the KSDISPATCH_TABLE at FsContext, which is defaulted to NULL within the kernel, at the FILE_OBJECT's creation stage. Thus the trick seems obvious, user-mode processes can allocate memory at 0x00000000 with no restriction, so finally we are controlling the pointer that is being dereferencing. Exploit it is a trivial task:

1. Allocate user-mode memory at 0
2. Set a fake FsContext pointer at 0 pointing to any other user-mode memory address.
3. Copy into the desired address our fake KSDISPATCH_TABLE
4. Issue a Write,Read... operation in the vulnerable device.

But wait, things are not so simple. WDM Drivers that binds to either PortCls.sys or Ks.sys through the proper supplied API will get their devices correctly initialized. The real problem occurs when the WDM driver creates additional devices since then all of them share a unique driver object which, in this particular case, may turn out into a potential vulnerability.

At this point, we have covered just one piece of the puzzle. Perhaps the more simple part. In the second stage of this paper we are going to explain the cases on which a WDM Driver may need to create additional devices and finally how an apparently safe driver turns out into a vulnerable one.

3. Second Stage

We have seen the root of the issue. Once these previous concepts have been assimilated we are in disposition to dig deeply and bring to the light those common driver programming methods that makes a driver suitable for being abused.

Devices whose purpose is not directly related with the Kernel Streaming API (KSA from now on) can be considered as “stranger” devices . We may define this term more globally as a device that has not been created or registered through the KSA so the KSA is not aware of it in any manner. These “stranger“ devices share the same driver object that the other KSA-related devices so the default KSA's IRP handlers pose a potential vulnerability.

Scenario 1

One common practice in driver programming is to create devices that acts as an interface to user-mode applications. Nothing new but let's think for a while about the new situationwe face while writing WDM audio drivers. We have initialized our driver object and *n* devices through the KSA. It installs several special IRP handlers which are fully, although **uniquely,** compliant with the KSA. If we create additional device(s) for our user-mode service we will be creating a potential attack vector as well.

Scenario 2

This one is the most interesting. Some relatively-old soundcards include a Gameport embedded. The Game Port connector used to be used by joysticks and that type of peripherals. Nowadays, USB is preferred for almost every modern gaming hardware so Microsoft has discontinued Game port support in Windows Vista. However , it still maintains a “user-friendly incompatibility” in order to avoid being rude with older hardware.

Vista gameport.inf

```
; gameport.inf - Hooks up known gameports with a NULL service.
;               Displays a name so the user can understand the device
;               is not supported out of the box.
; This avoids the '!' in device manager.
;
; Copyright (C) Microsoft Corporation.  All rights reserved.
```

WDM means stability, backward-compatibility and uniform architecture. This is the reason by which a driver that was coded more than 8 years ago can still be working in a Windows version for which it was not designed.

However, some WDM drivers may face now a different scenario while running on Microsoft Windows Vista. The Game port is no longer supported so this leads to important changes, not in the way the driver behaves but in the interfaces exposed by the driver.

Let's see the schema below to clarify this issue.

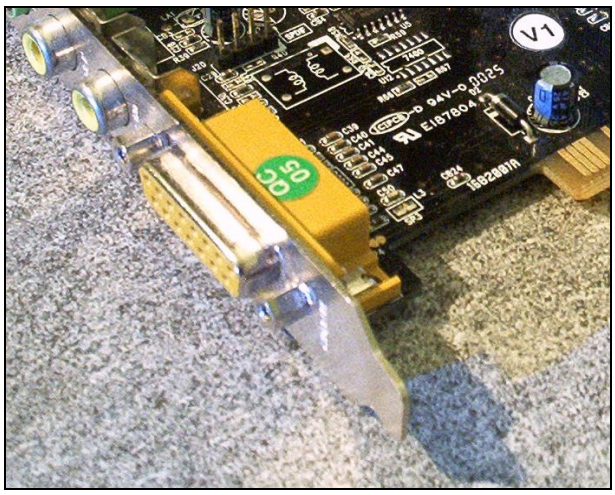

Image 2. Gameport Connector . Wikipedia.

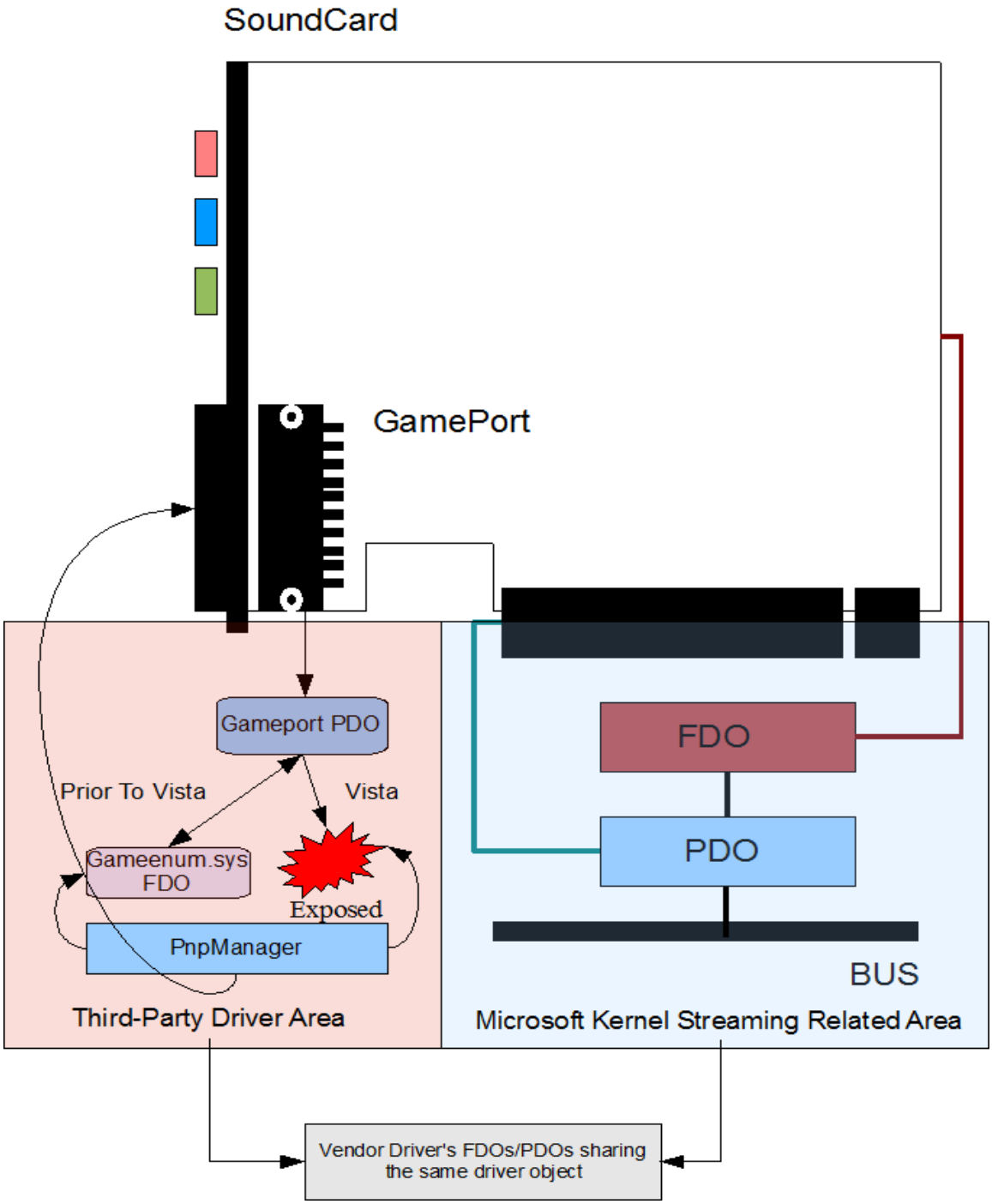

Image 3. Devices commonly exposed by a SoundCard/Driver

Within the blued “Microsoft Kernel Streaming Related Area” shape we can find all those device(s) the OS creates to identify the SoundCard, moreover we see the FDO(s) a driver will create through the KSA for interfacing with these PDOs. We assume that these devices are properly initialized since there is no reason for not doing so (a driver incorrectly binded to the KS simply will not work).

On the other hand , the “Third-Party Driver Area” comprises of those “stranger” devices we have been talking about previously. The startup sequence would be as follows:

In Versions prior to Vista

1. The PnP Manager issues a Query Relations request to our Driver that should create the GamePort PDO upon receiving this request.
2. The PnP loads gameenum.sys, creating an unnamed FDO for interfacing with the Gameport PDO , this FDO is attached to the stack. Thus, the access (its original IRPs handlers we mean) to the PDO from user-mode is not possible, despite the fact the PDO could be a named device.

In Windows Vista

1. The PnP Manager issues a Query Relations request to our Driver that will create the GamePort PDO upon receiving this request.
2. The PnP does not creates any FDO for interfacing with the Gameport PDO since the Game Port is not supported . Therefore, the PDO remains exposed to user-mode applications.
3. If the developer has not implemented the logic for routing correctly the IRPs sent to this device, we can reach the installed KSA's IRP handlers that will dereference our controlled FsContext pointer leading to arbitrary code execution in the Kernel.
4. We have fun.

All the stuff above is just a theoretical approach, but now we are going to see the second scenario in action.

The es1371mp.sys issue

This driver was created by CreativeLabs in the late 90's for widely extended “Ensoniq PCI 1371” based Sound Cards. Even today this driver is actively used since several VMware products emulate this soundcard. In fact, the driver is automatically installed through Microsoft Windows Update. We are talking about a WDM driver with near ten years of life...and it works. Amazing. Unfortunately, Creative is no longer supporting neither the hardware nor the driver so don't expect a patch.

Let's begin.

Binding to the PortCls.sys:

module: es1371mp.sys

```
1. PAGE:00014CE8 start           proc near
2. PAGE:00014CE8
3. PAGE:00014CE8 arg_0   = dword ptr  4
4. PAGE:00014CE8 arg_4   = dword ptr  8
5. PAGE:00014CE8
6. PAGE:00014CE8           push    ebx
7. PAGE:00014CE9           xor     ebx, ebx
8. [...]
9. PAGE:00014D2C           mov     esi, [esp+8+arg_0]
10.PAGE:00014D30           push    offset sub_14CCC ; AddDevice
11.PAGE:00014D35           push    [esp+0Ch+arg_4]
12.PAGE:00014D39           push    esi
13.PAGE:00014D3A           call    PcInitializeAdapterDriver
```

```
14.PAGE:00014D3F           cmp     eax, ebx
15.PAGE:00014D41           mov     dword ptr [esi+34h], offset sub_14780
16.PAGE:00014D48           jl      short loc_14D7A
17.PAGE:00014D4A           mov     dword ptr [esi+38h], offset sub_1803C
   ;overrides IRP_MJ_CREATE
18.PAGE:00014D51           mov     dword ptr [esi+40h], offset sub_1803C ;
   overrides IRP_MJ_CLOSE
19.PAGE:00014D58           mov     dword ptr [esi+0A4h], offset
   sub_17FC0;overrides IRP_MJ_PNP
20.PAGE:00014D62           mov     dword ptr [esi+90h], offset sub_17FE4; ...
21.PAGE:00014D6C           mov     dword ptr [esi+70h], offset
   sub_18072;overrides IRP_MJ_DEVICE_CONTROL
22.PAGE:00014D73           mov     dword ptr [esi+74h], offset sub_180AE;...
```

The driver is overriding several default KSA's IRP handlers, later on we'll see why.

Let's take a look at the new DispatchCreate (line 17)

module: es1371mp.sys

```
1. PAGE:0001803C sub_1803C       proc near               ; DATA XREF: start
   +62o
2. PAGE:0001803C                                         ; start+69o
3. PAGE:0001803C
4. PAGE:0001803C arg_0           = dword ptr  4
5. PAGE:0001803C Irp             = dword ptr  8
6. PAGE:0001803C
7. PAGE:0001803C                 mov     eax, [esp+arg_0]
8. PAGE:00018040                 mov     ecx, [eax+28h]
9. PAGE:00018043                 push    esi
10.PAGE:00018044                 xor     esi, esi
11.PAGE:00018046                 cmp     byte ptr [ecx+111h], 0 ;
   Discriminate  device by checking certain flag within the DeviceExtension
12.PAGE:0001804D                 jz      short loc_18060
13.PAGE:0001804F                 mov     ecx, [esp+4+Irp] ; Irp
14.PAGE:00018053                 and     [ecx+18h], esi
15.PAGE:00018056                 xor     dl, dl          ; PriorityBoost
16.PAGE:00018058                 call    ds:IofCompleteRequest
17.PAGE:0001805E                 jmp     short loc_1806C
18.PAGE:00018060 ;
   ---------------------------------------------------------------------------
   -
19.PAGE:00018060
20.PAGE:00018060 loc_18060:                              ; CODE XREF:
   sub_1803C+11j
21.PAGE:00018060                 push    [esp+4+Irp]
22.PAGE:00018064                 push    eax
23.PAGE:00018065
24.PAGE:00018065 loc_18065:
25.PAGE:00018065                 call    PcDispatchIrp
26.PAGE:0001806A                 mov     esi, eax
27.PAGE:0001806C
28.PAGE:0001806C loc_1806C:                              ; CODE XREF:
   sub_1803C+22j
29.PAGE:0001806C                 mov     eax, esi
30.PAGE:0001806E                 pop     esi
31.PAGE:0001806F                 retn    8
32.PAGE:0001806F sub_1803C       endp
```

We see how the routine discriminates between the upcoming GamePort PDO and the KSA device by checking a flag at the DeviceExtension. Thus, if the device receiving the IRP_MJ_CREATE is the KSA's device the function routes it by calling PortCls!PcDispatchIrp which, at certain point, will initialize the FsContext. Otherwise, the IRP is completed.

Now, time to EsDispatchPnp (Line 19)

```
module: es1371mp.sys
PAGE:00017FC0 sub_17FC0       proc near               ; DATA XREF: start+70o
PAGE:00017FC0
PAGE:00017FC0 arg_0           = dword ptr  4
PAGE:00017FC0 Irp             = dword ptr  8
PAGE:00017FC0
PAGE:00017FC0                 mov     eax, [esp+arg_0]
PAGE:00017FC4                 mov     ecx, [eax+28h]
PAGE:00017FC7                 cmp     byte ptr [ecx+111h], 0 ; Discriminates
Device by checking certain offset within the DeviceExtension
PAGE:00017FCE                 push    [esp+Irp]       ; Irp
PAGE:00017FD2                 push    eax             ; int
PAGE:00017FD3                 jz      short loc_17FDC
PAGE:00017FD5                 call    sub_188F2
PAGE:00017FDA                 jmp     short locret_17FE1
PAGE:00017FDC ;
---------------------------------------------------------------------------
PAGE:00017FDC
PAGE:00017FDC loc_17FDC:                              ; CODE XREF: sub_17FC0+13j
PAGE:00017FDC                 call    sub_18314
PAGE:00017FE1
PAGE:00017FE1 locret_17FE1:                           ; CODE XREF: sub_17FC0+1Aj
PAGE:00017FE1                 retn    8
PAGE:00017FE1 sub_17FC0       endp
```

The same implementation, it checks the device that is receiving the IRP and routes it. Digging into the handler for the IRP_MJ_PNP we find out where the PDO is created

```
PAGE:00018314 sub_18314       proc near               ; CODE XREF:
sub_17FC0:loc_17FDCp
PAGE:00018314
PAGE:00018314 arg_0           = dword ptr  8
PAGE:00018314 arg_4           = dword ptr  0Ch
PAGE:00018314
PAGE:00018314                 push    ebp
PAGE:00018315                 mov     ebp, esp
PAGE:00018317                 push    esi
PAGE:00018318                 mov     esi, [ebp+arg_4]
PAGE:0001831B                 mov     eax, [esi+60h]
PAGE:0001831E                 movzx   ecx, byte ptr [eax+1] ;MinorFunction
IRP_MN_QUERY_DEVICE_RELATIONS == 7
PAGE:00018322                 sub     ecx, 0
PAGE:00018325                 jz      short loc_18370
PAGE:00018327                 dec     ecx
PAGE:00018328                 dec     ecx
PAGE:00018329                 jz      short loc_18365
PAGE:0001832B                 sub     ecx, 5
PAGE:0001832E                 jz      short loc_1835A

[...]

PAGE:0001835A loc_1835A:                              ; CODE XREF: sub_18314+1Aj
PAGE:0001835A                 push    esi
PAGE:0001835B                 push    [ebp+arg_0]
PAGE:0001835E                 call    sub_181AA

[...]
PAGE:00018110 sub_18110       proc near             ; CODE XREF: sub_181AA+45p
PAGE:00018110
```

```
PAGE:00018110 var_8           = dword ptr -8
PAGE:00018110 var_4           = dword ptr -4
PAGE:00018110 arg_0           = dword ptr  8
PAGE:00018110
[...]
PAGE:0001811E                 push    ebx             ; DeviceObject
PAGE:0001811F                 push    1               ; Exclusive
PAGE:00018121                 push    80h             ; DeviceCharacteristics
PAGE:00018126                 push    2Ah             ; DeviceType
PAGE:00018128                 push    0               ; DeviceName
PAGE:0001812A                 push    180h            ; DeviceExtensionSize
PAGE:0001812F                 push    dword ptr [eax+8] ; DriverObject
PAGE:00018132                 mov     [ebp+var_4], ecx
PAGE:00018135                 call    ds:IoCreateDevice
PAGE:0001813B                 test    eax, eax
[...]
PAGE:00018158                 mov     [esi+164h], eax
PAGE:0001815E                 mov     byte ptr [esi+111h], 1  ; Sets Device Flag
```

Ok, all is up and running. This scenario has been working pretty well for years since you could not access the PDO. Nevertheless, in Vista the PDO is now exposed so the things have changed. This PDO shares its driver object with the KSA's Device and since the default IRP handlers Ks!DispatchCreate (ks.sys) and ks!DispatchRead has not been overridden after calling PcInitializeAdapterDriver, by issuing a IRP_MJ_WRITE (WriteFile) or IRP_MJ_READ (ReadFile) in the PDO we'll reach the vulnerable scenario we talked about in the first few pages.

The final outcome is that even guest users can elevate privileges to SYSTEM on affected installations.

So...

Where is the vulnerability?

In this paper we have just presented the facts supported by the technical details everyone can verify. You decide where the vulnerability is and who could be blamed for it (if any). Following a responsible policy disclosure Microsoft and VMware were contacted . They showed no interest.

How many drivers are affected?

We cannot say how many drivers are affected out there, but we strongly think there are several affected. Check your computer and if you want, contact us. The code presented in this paper belongs to the following driver.

Creative Ensoniq PCI ES1371 WDM Driver
es1371mp.sys
v 5.1.3612.0
Only vulnerable when it is running on Microsoft Windows Vista.

What's the solution?

From a developer standpoint the solution is not too complex, if you are writing a WDM driver that needs to create “stranger” devices you have to override every IRP handler included in the KSDISPATCH_TABLE. On the other side, Microsoft may avoid all these issues by checking for a NULL FsContext pointer before dereferencing it.

4. Conclusion

Writing secure drivers (secure code really) is not an easy task, there are dozens of important concepts involved, moreover a strong knowledge of the internals of the OS you are programming for is highly recommended. There is a method for modeling risks in complex systems known as the "Swiss Cheese Theory". This model is widely used in Aeronautical Industry and is also suitable for analyzing risk factors within the IT security Industry. Let's imagine several slices of Swiss Cheese, with all those tiny holes, each of these slices is a layer that is potentially avoiding that the threat can go forward through the holes, finally reaching the last stage of system. If all the layers fail, the whole system gets compromised and you may face an airplane crashing, a building collapsing or an attacker taking the control of your network. This paper is the story of what happens when all those "cheese" layers fails...

5. References

http://kartoffel.reversemode.com/downloads.php

Ksdispatch_plugin.zip
Exploit for es1371mp.sys + WDM Audio Drivers checker.

Reversemode
Advanced Reverse Engineering Services
contact (at) reversemode (dot) com
www.reversemode.com