NDI5aster
Privilege Escalation through NDIS 5.x Filter Intermediate Drivers

KYRIAKOS (kyREcon) ECONOMOU

CONTENTS

ABSTRACT

The Network Driver Interface Specification (NDIS) [11] provides a programming interface specification that facilitates from the network driver architecture perspective the communication between a protocol driver and the underlying network adapter. In Windows OS the so called "NDIS wrapper" (implemented in the *Ndis.sys*) provides a programming layer of communication between network protocols (TCP/IP) and all the underlying NDIS device drivers so that the implementation of high-level protocol components are independent of the network adapter itself. During vulnerability research from a local security perspective that was performed over several software firewall products designed for Windows XP and Windows Server 2003 (R2 included), an issue during the loading and initialization of one of the OS NDIS protocol drivers was identified; specifically the 'Remote Access and Routing Driver' called *wanarp.sys*. This issue can be exploited through various NDIS 5.x filter intermediate drivers [4] that provide the firewall functionality of several security related products. The resulting impact is vertical privilege escalation which allows a local attacker to execute code with kernel privileges from any account type, thus completely compromising the affected host.

1 INTRODUCTION

Security software should provide security. This is what makes research over those bits and bytes slightly more interesting than researching on other software types. The impact is not necessarily greater, since other more mainstream applications might be more widely used and commonly installed, but the word 'security' is what makes them really attractive to us. On the other hand, we have Windows XP, which was recently abandoned by Microsoft in terms of security patching which means all the 'goodies' that we can find there, will also probably stay there forever. Unless Microsoft decides to jump back and apply new patches, we can safely say that "what happens in XP stays in XP".

However, Windows Server 2003, which is also affected by the examined issue, was still officially supported by Microsoft at the time of writing this paper. Although, XP operating system is not supported anymore by the vendor it is still quite widely used internally in many companies, and especially Windows Server 2003 R2. These hosts might run important infrastructure software that might not be supported anymore by its vendor. At the same time the migration to a newer platform and finding the right software to rebuilt those systems with the same capabilities might be extremely time and money consuming. In a fair attempt to harden their security, system administrators will install some security software on them. This quite often implies installing some AV security suite that provides malware detection and elimination, as well as some extra firewall capabilities.

Based on the aforementioned facts, this research aims to bring some awareness about a well hidden for years issue that even though is not really a bug by definition, it can be exploited through NDIS 5.x network intermediate drivers used by software firewalls to filter network packets [4]. Upon exploitation, it allows a local attacker to elevate his privileges and obtain complete access on the compromised host. This can later lead to a total compromise of the network infrastructure through common post-exploitation techniques, such as obtaining important cached credentials through hash dumping or live credentials residing in memory.

2 NDIS 5.X

The NDIS acronym refers to the Network Driver Interface Specification [11] which defines the way network protocols communicate with the underlying network adapters. It provides a set of routines that allow the network drivers that implement protocols to communicate with the NDIS wrapper instead of directly accessing the Network Interface Card (NIC) NDIS driver as seen in Figure 1 on the following page.

This allows the protocol implementation to be independent from the NDIS miniport device drivers. The research focused on NDIS 5.x which is the major version of the NDIS wrapper that was primarily introduced in Windows 2000 (NDIS 5.0) and later improved in Windows XP (NDIS 5.1) kernel based operating systems.

Before going into the details of the discovered vulnerability, the way it can be exploited, and under which circumstances, it is important to provide an insight of the types of NDIS 5.x drivers that are available.

2.1 Protocol Drivers

These drivers are used to implement network protocols [6]. They are located at the highest position in the NDIS hierarchy of drivers and they are used as the lowest-level drivers when implementing a transport driver and the associated protocol stack such as the TCP/IP stack. These drivers also need to implement an interface in order

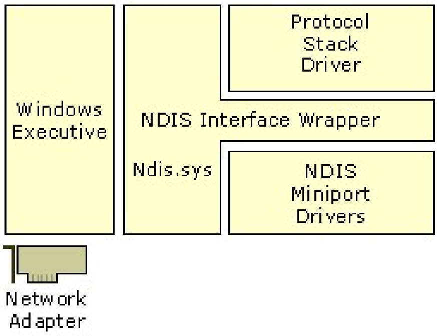


Figure 1: NDIS Wrapper Architecture

to receive incoming packets from the next driver in the stack bellow them and in case of a transport protocol driver, the driver needs to transfer the incoming data to the appropriate application as well. At its lower edge, a protocol driver provides an interface of communication with an underlying intermediate driver, if there is one, or with a miniport driver that is the one that communicates with the physical device. At its upper edge, a protocol driver interfaces with a higher-level driver which makes part of the protocol stack.

Protocol drivers import *NdisXxx* functions ('Xxx' is being used throughout this paper as a function name placeholder) that are used to perform various operations, such as sending packets, setting information that needs to be maintained by lower-level drivers, as well as making use of specific services provided by the operating system. Furthermore, protocol drivers also export *ProtocolXxx* functions that the NDIS wrapper uses to perform operations on behalf of lower-level drivers. These might be indicating the receiving of packets, retrieving information about the status of a lower-level drivers and in general allowing NDIS to communicate with the protocol driver.

2.2 Miniport Drivers

These are the NDIS device drivers that communicate with the network adapters (NIC devices) at their lower edge, while at their upper edge they provide an interface of the lower edge of protocol drivers [10]. Miniport and protocol drivers are essential components of an NDIS driver stack. Figure 2 demonstrates from a high-level perspective how these relate to each other and with the NDIS wrapper [3].

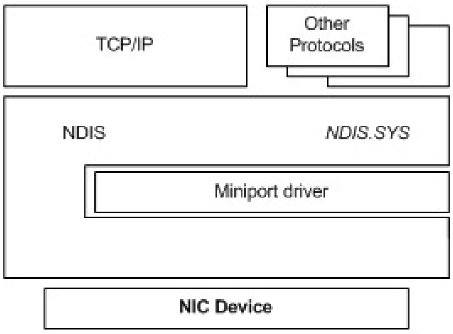


Figure 2: NDIS Driver Stack

2.3 Intermediate Drivers

As their name suggests, these NDIS drivers are located between the protocol drivers and the miniport drivers [3]. These drivers are not essential components of an NDIS driver stack unless there is a need for parsing, filtering, logging for security or any other purpose that requires some sort of processing of the data that travels between the higher level protocol drivers and the lower miniport drivers that control physical devices. In order to achieve this purpose, intermediate drivers expose a protocol driver interface on their upper edge and miniport driver interface at their lower edge which in this case is called virtual miniport. It is called 'virtual' because it does not actually control a physical device. Instead, it has to interface with the underlying miniport driver which is the one that actually controls the NIC device. Figure 3 shows an example of an NDIS driver stack where an intermediate driver is loaded in between the protocol driver and the miniport driver[3]. However, it is possible that more than one intermediate drivers are loaded at the same time in an NDIS driver stack.

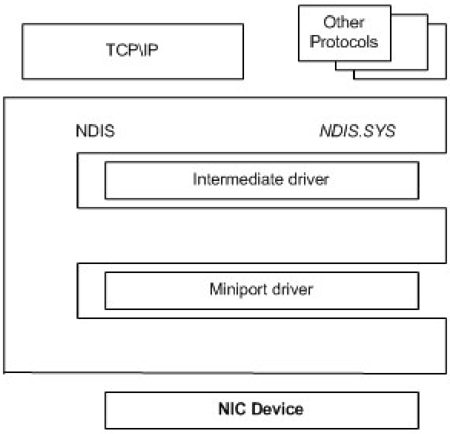


Figure 3: NDIS Driver Stack with an Intermediate Driver

It is important to mention that in NDIS 5.x more than one miniport drivers can be bound to lower protocol edge of an intermediate driver[11]. In that case, the intermediate driver needs to expose an equal amount of virtual miniports on its upper edge so that higher-level drivers or intermediate drivers can interface with them via their lower protocol edge.

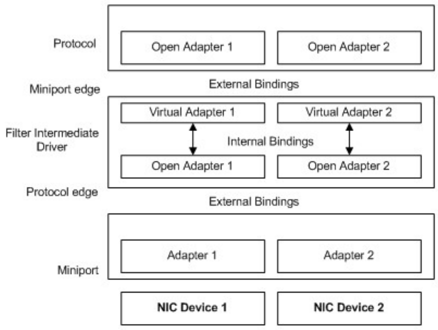


Figure 4: One-to-one relationship between miniport drivers and virtual miniports

There are two types of this category of NDIS drivers. The *NDIS filter intermediate drivers*, and the *MUX intermediate drivers*. The former ones are those that are used in many firewall, VPN, and other networking related software products built over the NDIS 5.x for the Windows XP and Windows Server 2003 operating systems.

2.3.1 *External & Internal Bindings*

As we have already discussed, intermediate drivers can bind with other drivers or other intermediate drivers. These bindings [3] are controlled by the NDIS wrapper and for this reason they are called external bindings. However, intermediate drivers bind their own protocol edge and virtual miniport edge internally. These are called internal bindings because they are not controlled by NDIS and their implementation can be completely custom and vendor specific. Figure 5 demonstrates the internal binding between the virtual miniport and the intermediate driver's protocol interface. We can also observe this characteristic in Figure 4 on the preceding page .

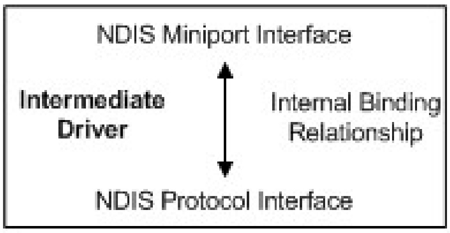


Figure 5: Internal Binding

Now that we have finished with a short overview of the NDIS 5.x drivers, we will proceed with some additional information about protocol and intermediate drivers which is necessary to mention in the context of this research in order to understand later the root of the issue that motivated the writing of this paper.

3 REGISTERING AN NDIS 5.X PROTOCOL DRIVER

On loading, an NDIS protocol driver needs to register its *ProtocolXxx* functions by calling the *NdisRegisterProtocol* function (see figure 6) from inside its *DriverEntry* which is basically the standard entry point function name for a kernel mode driver that is recognized by the loader [6]. The handle that will be stored as *NdisProtocolHandle* after a successful call to the aforementioned function must be preserved by the driver since it will be later needed in other calls to NDIS functions.

```
VOID NdisRegisterProtocol(
  _Out_  PNDIS_STATUS Status,
  _Out_  PNDIS_HANDLE NdisProtocolHandle,
  _In_   PNDIS_PROTOCOL_CHARACTERISTICS ProtocolCharacteristics,
  _In_   UINT CharacteristicsLength
);
```

Figure 6: NdisRegisterProtocol

However, before calling the *NdisRegisterProtocol* function, the driver needs to zero-initialize the `NDIS_PROTOCOL_CHARACTERISTICS` structure in order to ensure that any unused members are set to NULL. Even though following this good practice can later help a caller to check if a function pointer in this structure is initialized or not, this is not enough as demonstrated later in this paper. Once the structure has been zero-initialized, the driver also needs to set the NDIS version with which the protocol is compatible. Finally, the driver needs to set accordingly the function pointers of the necessary and optional *ProtocolXxx* functions that the driver exports. Once this final step is done, the driver is ready to call the *NdisRegisterProtocol* function.

Figure 7 shows the `NDIS_PROTOCOL_CHARACTERISTICS` structure definition.

```
typedef struct _NDIS_PROTOCOL_CHARACTERISTICS {
    UCHAR MajorNdisVersion;
    UCHAR MinorNdisVersion;
    UINT Reserved;
    OPEN_ADAPTER_COMPLETE_HANDLER OpenAdapterCompleteHandler;
    CLOSE_ADAPTER_COMPLETE_HANDLER CloseAdapterCompleteHandler;
    SEND_COMPLETE_HANDLER SendCompleteHandler;
    TRANSFER_DATA_COMPLETE_HANDLER TransferDataCompleteHandler;
    RESET_COMPLETE_HANDLER ResetCompleteHandler;
    REQUEST_COMPLETE_HANDLER RequestCompleteHandler;
    RECEIVE_HANDLER ReceiveHandler;
    RECEIVE_COMPLETE_HANDLER ReceiveCompleteHandler;
    STATUS_HANDLER StatusHandler;
    STATUS_COMPLETE_HANDLER StatusCompleteHandler;
    NDIS_STRING Name;
//
// MajorNdisVersion must be set to 0x04 or 0x05
// with any of the following members.
//
    RECEIVE_PACKET_HANDLER ReceivePacketHandler;
    BIND_HANDLER BindAdapterHandler;
    UNBIND_HANDLER UnbindAdapterHandler;
    PNP_EVENT_HANDLER PnPEventHandler;
    UNLOAD_PROTOCOL_HANDLER UnloadHandler;
//
// MajorNdisVersion must be set to 0x05
// with any of the following members.
//
    CO_SEND_COMPLETE_HANDLER CoSendCompleteHandler;
    CO_STATUS_HANDLER CoStatusHandler;
    CO_RECEIVE_PACKET_HANDLER CoReceivePacketHandler;
    CO_AF_REGISTER_NOTIFY_HANDLER CoAfRegisterNotifyHandler;
} NDIS_PROTOCOL_CHARACTERISTICS, *PNDIS_PROTOCOL_CHARACTERISTICS;
```

Figure 7: NDIS_PROTOCOL_CHARACTERISTICS structure

4 REGISTERING AN NDIS 5.X INTERMEDIATE DRIVER

During initialization, an NDIS intermediate driver needs also to perform a few calls to some NDIS functions in the context of its *DriverEntry* function in order to register its MiniportXxx functions and its *ProtocolXxx* functions in case it has to bind to a lower-level NDIS driver. As a first step, the intermediate driver needs to call *NdisMInitializeWrapper* in order to notify NDIS that a new miniport driver is currently initializing [8].

```
VOID NdisMInitializeWrapper(
  _Out_  PNDIS_HANDLE NdisWrapperHandle,
  _In_   PVOID SystemSpecific1,
  _In_   PVOID SystemSpecific2,
  _In_   PVOID SystemSpecific3
);
```

Figure 8: NdisMInitializeWrapper

The handle stored in *NdisWrapperHandle* will be later used as parameter to other calls of NDIS functions (Figure 8 on the previous page). Assuming that this first action was successful, the intermediate driver will subsequently call *NdisIMRegisterLayeredMiniport* through which will register with NDIS the entry points of the *MiniportXxx* functions that it exports.

```
NDIS_STATUS NdisIMRegisterLayeredMiniport(
  _In_    NDIS_HANDLE NdisWrapperHandle,
  _In_    PNDIS_MINIPORT_CHARACTERISTICS MiniportCharacteristics,
  _In_    UINT CharacteristicsLength,
  _Out_   PNDIS_HANDLE DriverHandle
);
```

Figure 9: NdisIMRegisterLayeredMiniport

If the driver has to bind to a lower level NDIS driver, then it will also call *NdisRegisterProtocol* in order to register the entry points of the *ProtocolXxx* functions that it exports [9].

```
VOID NdisRegisterProtocol(
  _Out_   PNDIS_STATUS Status,
  _Out_   PNDIS_HANDLE NdisProtocolHandle,
  _In_    PNDIS_PROTOCOL_CHARACTERISTICS ProtocolCharacteristics,
  _In_    UINT CharacteristicsLength
);
```

Figure 10: NdisRegisterProtocol

There is a particular interest in the third parameter of this function which is a pointer to a `NDIS_PROTOCOL_CHARACTERISTICS` structure (see figure 7 on the preceding page), which as you can see stores the pointers to functions that need to handle certain events. The `RECEIVE_COMPLETE_HANDLER` member of this structure is of a particular interest since it is root cause of the issue we are about to examine. Finally, the intermediate driver needs to call *NdisIMAssociateMiniport*. This is done in order to inform NDIS that the specified protocol and miniport interfaces, referenced by the handles passed as parameters to this function (see figure 11) belong to the same intermediate driver.

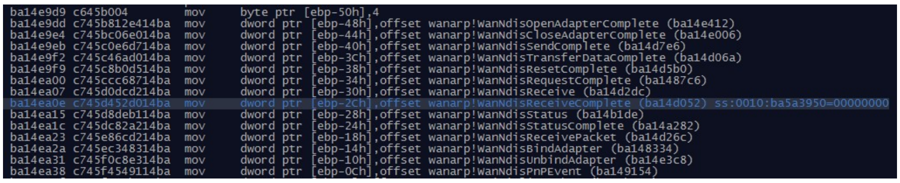
```
VOID NdisIMAssociateMiniport(
  _In_   NDIS_HANDLE DriverHandle,
  _In_   NDIS_HANDLE ProtocolHandle
);
```

Figure 11: NdisIMAssociateMiniport

To be more specific, the *DriverHandle* is the handle to the miniport interface returned by *NdisIMRegisterLayeredMiniport*, and the *ProtocolHandle* is the one returned by *NdisRegisterProtocol* function. In cases where the intermediate driver is bound to more than one miniport drivers (see section 2.3 on page 5), then it has to call *NdisIMInitializeDeviceInstanceEx* for every virtual NIC that makes available so that higher level protocol drivers can bind to it and send network requests.

5 WANARP.SYS - PROTOCOL REGISTRATION

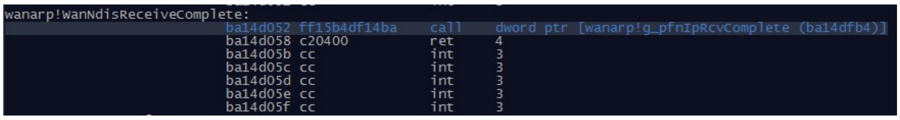
This NDIS protocol driver of Windows OS is described as the '*Remote Access and Routing ARP Driver*'. To be more specific, the *wanarp.sys* (v5.1.2600.5512) file under examination is part of an a XP SP3 32-bit installation. The root cause of the issue that we are about to exploit is located in the registration stage of the protocol itself which in this case occurs inside the *wanarp!WanpInitializeNdis* function. During this stage, a protocol driver needs to initialize a `NDIS_PROTOCOL_CHARACTERISTICS` structure (see figure 7 on page 7) with pointers to the ProtocolXxx functions that it exports. This structure is correctly zero-initialized and then valid pointers are stored to the necessary members for this protocol. However, something is about to go wrong. Really wrong! Let's take a look at the following figure.



```
ba14e9d9 c645b004        mov     byte ptr [ebp-50h],4
ba14e9dd c745b812e414ba  mov     dword ptr [ebp-48h],offset wanarp!WanNdisOpenAdapterComplete (ba14e412)
ba14e9e4 c745bc06e014ba  mov     dword ptr [ebp-44h],offset wanarp!WanNdisCloseAdapterComplete (ba14e006)
ba14e9eb c745c0e6d714ba  mov     dword ptr [ebp-40h],offset wanarp!WanNdisSendComplete (ba14d7e6)
ba14e9f2 c745c46ad014ba  mov     dword ptr [ebp-3Ch],offset wanarp!WanNdisTransferDataComplete (ba14d06a)
ba14e9f9 c745c8b0d514ba  mov     dword ptr [ebp-38h],offset wanarp!WanNdisResetComplete (ba14d5b0)
ba14ea00 c745ccc68714ba  mov     dword ptr [ebp-34h],offset wanarp!WanNdisRequestComplete (ba1487c6)
ba14ea07 c745d0dcd214ba  mov     dword ptr [ebp-30h],offset wanarp!WanNdisReceive (ba14d2dc)
ba14ea0e c745d452d014ba  mov     dword ptr [ebp-2Ch],offset wanarp!WanNdisReceiveComplete (ba14d052) ss:0010:ba5a3950=00000000
ba14ea15 c745d8deb114ba  mov     dword ptr [ebp-28h],offset wanarp!WanNdisStatus (ba14b1de)
ba14ea1c c745dc82a214ba  mov     dword ptr [ebp-24h],offset wanarp!WanNdisStatusComplete (ba14a282)
ba14ea23 c745e86cd214ba  mov     dword ptr [ebp-18h],offset wanarp!WanNdisReceivePacket (ba14d26c)
ba14ea2a c745ec348314ba  mov     dword ptr [ebp-14h],offset wanarp!WanNdisBindAdapter (ba148334)
ba14ea31 c745f0c8e314ba  mov     dword ptr [ebp-10h],offset wanarp!WanNdisUnbindAdapter (ba14e3c8)
ba14ea38 c745f4549114ba  mov     dword ptr [ebp-0Ch],offset wanarp!WanNdisPnPEvent (ba149154)
```

Figure 12: wanarp.sys - Protocol Functions Registration

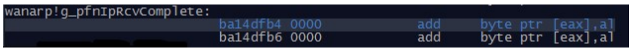
All these are valid pointers, however let's examine closer the highlighted pointer that is passed to the *`RECEIVE_COMPLETE_HANDLER`* member.



```
wanarp!WanNdisReceiveComplete:
                                ba14d052 ff15b4df14ba    call    dword ptr [wanarp!g_pfnIpRcvComplete (ba14dfb4)]
                                ba14d058 c20400          ret     4
                                ba14d05b cc              int     3
                                ba14d05c cc              int     3
                                ba14d05d cc              int     3
                                ba14d05e cc              int     3
                                ba14d05f cc              int     3
```

Figure 13: wanarp!WanNdisReceiveComplete

The pointer passed to the aforementioned member points to a legitimate function *WanNdisReceiveComplete*, (see figure 12) inside the *wanarp.sys* module. Notice that the only thing this function does, is basically to call a function through a pointer stored in *wanarp!g_pfnIpRcvComplete* dword (see figure 13) located in the *.data* section of the module at *RVA*[1]*: wanarp + 0x5FB4*.



```
wanarp!g_pfnIpRcvComplete:
                              ba14dfb4 0000           add     byte ptr [eax],al
                              ba14dfb6 0000           add     byte ptr [eax],al
```

Figure 14: wanarp!`g_pfnIpRcvComplete`

As we can see in figure 14, the *g_pfnIpRcvComplete* pointer is NULL which means that *WanNdisReceiveComplete* is basically performing a *Call 0x00000000*. In Windows XP and Server 2003 based systems allocating the NULL page is not a problem at all, but first we need to find a way to control a call to that function.

6 ESET - EPFWNDIS.SYS

This issue initially caught our attention while looking for vulnerabilities in the latest (at the time) ESET '*Smart Security*' product for Windows XP (SP3). Later on it was proved that their latest '*Endpoint Security*' product for Windows Server 2003 was also vulnerable to privilege escalation through the same attack type, along with other similar products from other vendors. Although, this is not a vulnerability caused by a programming error, we can categorize the fact that a driver allows us to trigger it as a *design error* that produces a '*trusted value vulnerability*' situation, as we are going

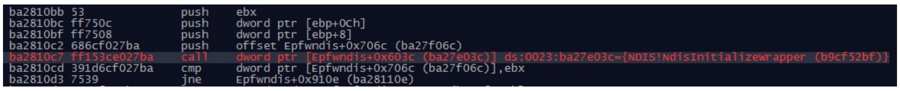
---

[1] Note: The reason why in some cases it is preferable to refer to relative virtual addresses (RVA) has to do with the fact that during the writing of this paper different instances of the OS during the drivers' loading stage have been examined. This means that a virtual address (VA) referring to the same location in a loaded module might change on each reboot, but the RVA will not. All of the RVAs mentioned are calculated using the image base of the corresponding module as a reference.

to see in detail. The analysis that follows is based on the *Epfwndis.sys v7.0.206.0* also known as '*ESET Personal Firewall NDIS filter*'.

6.1 Driver Initialization

As it has been discussed in section 4 on page 7, an NDIS intermediate driver needs to perform some necessary steps during initialization. We notice the call to *NdisInitializeWrapper*[2] at address *Epfwndis + 0x90C7* (see figure 15).



```
ba2810bb 53              push    ebx
ba2810bc ff750c          push    dword ptr [ebp+0Ch]
ba2810bf ff7508          push    dword ptr [ebp+8]
ba2810c2 686cf027ba      push    offset Epfwndis+0x706c (ba27f06c)
ba2810c7 ff153ce027ba    call    dword ptr [Epfwndis+0x603c (ba27e03c)] ds:0023:ba27e03c={NDIS!NdisInitializeWrapper (b9cf52bf)}
ba2810cd 391d6cf027ba    cmp     dword ptr [Epfwndis+0x706c (ba27f06c)],ebx
ba2810d3 7539            jne     Epfwndis+0x910e (ba28110e)
```

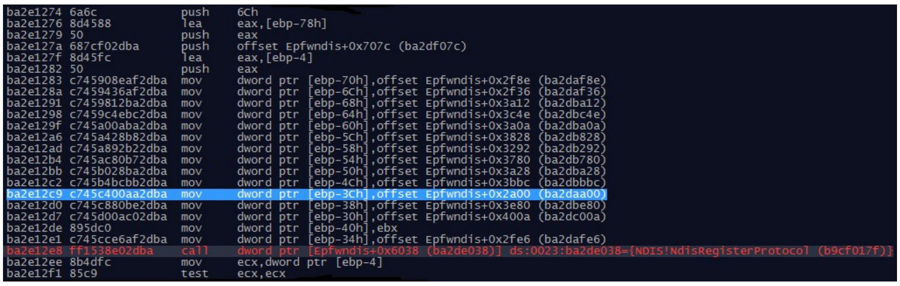
Figure 15: Epfwndis - Call NdisInitializeWrapper

The declaration of this function has been provided already (see figure 7 on page 7), so we know that the *NdisWrapperHandle* is going to be stored at *Epfwndis + 0x706C*. This handle is important since it is going to be needed later for other calls to NDIS functions. The next important step, and as expected from what we have already discussed (see section 4 on page 7), the ESET driver is going to make a call from address *Epfwndis + 0x91F6* to *NdisIMRegisterLayeredMiniport* (see figure 9 on page 8) which is necessary in order to register the entry points of the exported *MiniportXxx* functions, as shown in the figure that follows.

```
ba2b1167 8d850cffffff     lea     eax,[ebp-0F4h]
ba2b116d 6880f02aba       push    offset Epfwndis+0x7080 (ba2af080)
ba2b1172 6a7c             push    7Ch
ba2b1174 50               push    eax
ba2b1175 ff356cf02aba     push    dword ptr [Epfwndis+0x706c (ba2af06c)]
ba2b117b 66c7850cffffff0501 mov   word ptr [ebp-0F4h],105h
ba2b1184 c78528ffffff4e972aba mov dword ptr [ebp-0D8h],offset Epfwndis+0x174e (ba2a974e)
ba2b118e c78530ffffffae992aba mov dword ptr [ebp-0D0h],offset Epfwndis+0x19ae (ba2a99ae)
ba2b1198 c78540ffffff049d2aba mov dword ptr [ebp-0C0h],offset Epfwndis+0x1d04 (ba2a9d04)
ba2b11a2 899d38ffffff     mov     dword ptr [ebp-0C8h],ebx
ba2b11a8 c78544ffffffbe9e2aba mov dword ptr [ebp-0BCh],offset Epfwndis+0x1ebe (ba2a9ebe)
ba2b11b2 c78520ffffff12962aba mov dword ptr [ebp-0E0h],offset Epfwndis+0x1612 (ba2a9612)
ba2b11bc c7856cffffff5a952aba mov dword ptr [ebp-94h],offset Epfwndis+0x155a (ba2a955a)
ba2b11c6 c78570ffffffd2952aba mov dword ptr [ebp-90h],offset Epfwndis+0x15d2 (ba2a95d2)
ba2b11d0 c78574ffffff52952aba mov dword ptr [ebp-8Ch],offset Epfwndis+0x1552 (ba2a9552)
ba2b11da 899d14ffffff     mov     dword ptr [ebp-0ECh],ebx
ba2b11e0 899d48ffffff     mov     dword ptr [ebp-0B8h],ebx
ba2b11e6 899d3cffffff     mov     dword ptr [ebp-0C4h],ebx
ba2b11ec c7854cffffff1c9c2aba mov dword ptr [ebp-0B4h],offset Epfwndis+0x1c1c (ba2a9c1c)
ba2b11f6 ff1544e02aba     call    dword ptr [Epfwndis+0x6044 (ba2ae044)] ds:0023:ba2ae044={NDIS!NdisIMRegisterLayeredMiniport (b9cf5a8e)}
ba2b11fc 8bc8             mov     ecx,eax
ba2b11fe 894dfc           mov     dword ptr [ebp-4],ecx
```

Figure 16: Epfwndis - Call NdisIMRegisterLayeredMiniport

Once the previous step has be accomplished, the driver will now register its *ProtocolXxx* functions with a call to *NdisRegisterProtocol* from adress *Epfwndis + 0x92E8*.



```
ba2e1274 6a6c            push    6Ch
ba2e1276 8d4588          lea     eax,[ebp-78h]
ba2e1279 50              push    eax
ba2e127a 687cf02dba      push    offset Epfwndis+0x707c (ba2df07c)
ba2e127f 8d45fc          lea     eax,[ebp-4]
ba2e1282 50              push    eax
ba2e1283 c745908eaf2dba  mov     dword ptr [ebp-70h],offset Epfwndis+0x2f8e (ba2daf8e)
ba2e128a c7459436af2dba  mov     dword ptr [ebp-6Ch],offset Epfwndis+0x2f36 (ba2daf36)
ba2e1291 c7459812ba2dba  mov     dword ptr [ebp-68h],offset Epfwndis+0x3a12 (ba2dba12)
ba2e1298 c7459c4ebc2dba  mov     dword ptr [ebp-64h],offset Epfwndis+0x3c4e (ba2dbc4e)
ba2e129f c745a00aba2dba  mov     dword ptr [ebp-60h],offset Epfwndis+0x3a0a (ba2dba0a)
ba2e12a6 c745a428b82dba  mov     dword ptr [ebp-5Ch],offset Epfwndis+0x3828 (ba2db828)
ba2e12ad c745a892b22dba  mov     dword ptr [ebp-58h],offset Epfwndis+0x3292 (ba2db292)
ba2e12b4 c745ac80b72dba  mov     dword ptr [ebp-54h],offset Epfwndis+0x3780 (ba2db780)
ba2e12bb c745b028ba2dba  mov     dword ptr [ebp-50h],offset Epfwndis+0x3a28 (ba2dba28)
ba2e12c2 c745b4bcbb2dba  mov     dword ptr [ebp-4Ch],offset Epfwndis+0x3bbc (ba2dbbbc)
ba2e12c9 c745c400aa2dba  mov     dword ptr [ebp-3Ch],offset Epfwndis+0x2a00 (ba2daa00)
ba2e12d0 c745c880be2dba  mov     dword ptr [ebp-38h],offset Epfwndis+0x3e80 (ba2dbe80)
ba2e12d7 c745d00ac02dba  mov     dword ptr [ebp-30h],offset Epfwndis+0x400a (ba2dc00a)
ba2e12de 895dc0          mov     dword ptr [ebp-40h],ebx
ba2e12e1 c745cce6af2dba  mov     dword ptr [ebp-34h],offset Epfwndis+0x2fe6 (ba2dafe6)
ba2e12e8 ff1538e02dba    call    dword ptr [Epfwndis+0x6038 (ba2de038)] ds:0023:ba2de038={NDIS!NdisRegisterProtocol (b9cf017f)}
ba2e12ee 8b4dfc          mov     ecx,dword ptr [ebp-4]
ba2e12f1 85c9            test    ecx,ecx
```

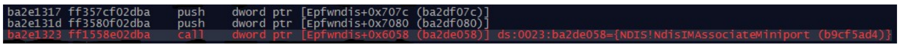
Figure 17: Epfwndis - Call NdisRegisterProtocol

In the figure above, the pointer stored in the *BindAdapterHandler* member of the NDIS_PROTOCOL_CHARACTERISTICS structure has been intentionally highlighted. This function will be later used as a callback by NDIS in order to bind the current

---

[2] Note: In this case the driver is calling *NdisInitializeWrapper* instead of *NdisMInitializeWrapper* (see figure 8 on page 7). According to MSDN, this is an obsolete function that only exists to support legacy NDIS v3.0 drivers and normally shouldn't be used for NDIS 4.0, NDIS 5.0 drivers and NDIS 3.0 miniport drivers [7].

driver to the underlying NIC drivers. This is called *ProtocolBindAdapter* function (see section 6.2) and it is used in order to support plug and play, hence it is called whenever a NIC where the protocol can bind itself becomes available. In this case the function that will handle this purpose on behalf of this ESET driver, is located at *Epfwndis + 0x2A00*. The next important function that is called is the *NdisIMAssociateMiniport* (see figure 18), which serves to inform NDIS that a specific protocol and miniport interfaces, they both belong to the same intermediate driver (see section 4 on page 7).

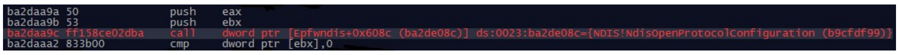


```
ba2e1317 ff357cf02dba    push    dword ptr [Epfwndis+0x707c (ba2df07c)]
ba2e131d ff3580f02dba    push    dword ptr [Epfwndis+0x7080 (ba2df080)]
ba2e1323 ff1558e02dba    call    dword ptr [Epfwndis+0x6058 (ba2de058)] ds:0023:ba2de058={NDIS!NdisIMAssociateMiniport (b9cf5ad4)}
```

Figure 18: Epfwndis - Call NdisIMAssociateMiniport

6.2 ProtocolBindAdapter

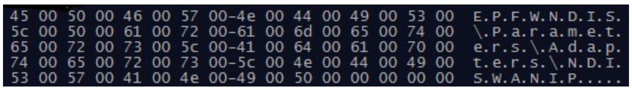
We are about to examine the steps that are taken during the binding between *Epfwndis* and other underlying NDIS drivers. As mentioned earlier, the amount of times that the *ProtocolBindAdapter* function is going to be called it depends also on the amount of existing active network adapters. The first important call is to *NdisOpenProtocolConfiguration* from address *Epfwndis + 0x2A9C* (see figure 19). This function returns a handle to the registry key where the per-adapter information of a protocol driver is stored.



```
ba2daa9a 50             push    eax
ba2daa9b 53             push    ebx
ba2daa9c ff158ce02dba   call    dword ptr [Epfwndis+0x608c (ba2de08c)] ds:0023:ba2de08c={NDIS!NdisOpenProtocolConfiguration (b9cfdf99)}
ba2daaa2 833b00         cmp     dword ptr [ebx],0
```

Figure 19: Epfwndis - Call NdisOpenProtocolConfiguration

The first underlying registered adapter that is going to bind to, is the *NDISWANIP* as shown in the following figure.



```
45 00 50 00 46 00 57 00-4e 00 44 00 49 00 53 00  E.P.F.W.N.D.I.S.
5c 00 50 00 61 00 72 00-61 00 6d 00 65 00 74 00  \.P.a.r.a.m.e.t.
65 00 72 00 73 00 5c 00-41 00 64 00 61 00 70 00  e.r.s.\.A.d.a.p.
74 00 65 00 72 00 73 00-5c 00 4e 00 44 00 49 00  t.e.r.s.\.N.D.I.
53 00 57 00 41 00 4e 00-49 00 50 00 00 00 00 00  S.W.A.N.I.P.....
```

Figure 20: Binding to NDISWANIP

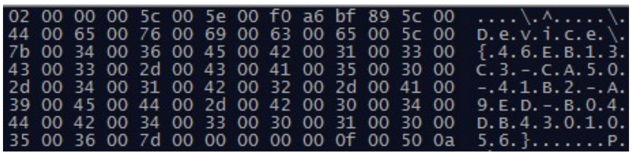
The retrieved handle from the previous call is going to be used immediately after, to call *NdisReadConfiguration* from address *Epfwndis + 0x2AC2*.

This is done in order to obtain the value of a named entry belonging to the previously opened registry key. The entry it is about to examine is called "*UpperBindings*". The data is returned to an NDIS_CONFIGURATION_PARAMETER structure which is defined in the following figure.

```
VOID NdisRegisterProtocol(
  _Out_  PNDIS_STATUS Status,
  _Out_  PNDIS_HANDLE NdisProtocolHandle,
  _In_   PNDIS_PROTOCOL_CHARACTERISTICS ProtocolCharacteristics,
  _In_   UINT CharacteristicsLength
);
```

Figure 21: `BNDIS_CONFIGURATION_PARAMETER` structure

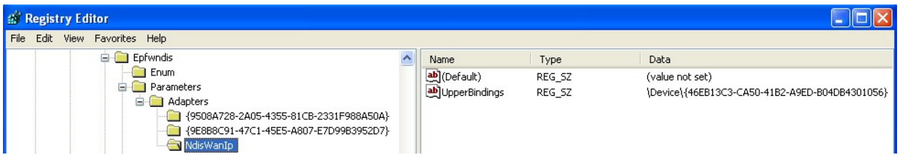
Figure 22 shows an example of the data returned in this case.



```
02 00 00 00 5c 00 5e 00 f0 a6 bf 89 5c 00  ....\.^.....\.
44 00 65 00 76 00 69 00 63 00 65 00 5c 00  D.e.v.i.c.e.\.
7b 00 34 00 36 00 45 00 42 00 31 00 33 00  {.4.6.E.B.1.3.
43 00 33 00 2d 00 43 00 41 00 35 00 30 00  C.3.-.C.A.5.0.
2d 00 34 00 31 00 42 00 32 00 2d 00 41 00  -.4.1.B.2.-.A.
39 00 45 00 44 00 2d 00 42 00 30 00 34 00  9.E.D.-.B.0.4.
44 00 42 00 34 00 33 00 30 00 31 00 30 00  D.B.4.3.0.1.0.
35 00 36 00 7d 00 00 00 00 00 0f 00 50 0a  5.6.}.......P.
```

Figure 22: Data returned through a call to NdisReadConfiguration

The device name identifier shown in the figure above refers to the virtual miniport instance created for the *NDISWANIP* (miniport driver) adapter. We can also verify this setting by looking at the registry.

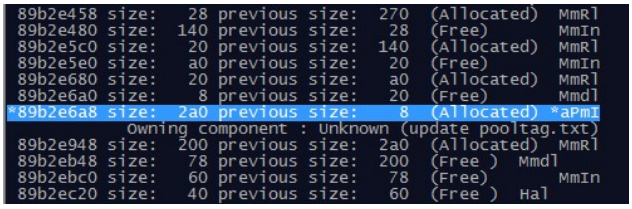


```
Epfwndis
  Enum
  Parameters
    Adapters
      {9508A728-2A05-4355-81CB-2331F988A50A}
      {9E8B8C91-47C1-45E5-A807-E7D99B3952D7}
      NdisWanIp
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
UpperBindings	REG_SZ	\Device\{46EB13C3-CA50-41B2-A9ED-B04DB4301056}

Figure 23: Virtual miniport identifier for NdisWanIp

As a next step, *Epfwndis* will call *NdisAllocateMemoryWithTag* in order to allocate some memory and save that information in a nonpaged tagged pool buffer. This is also helpful for us to know since we can use it at any point to find other memory blocks allocated with that specified third-party pool tag which is the "*aPmI*" (see figure 24).

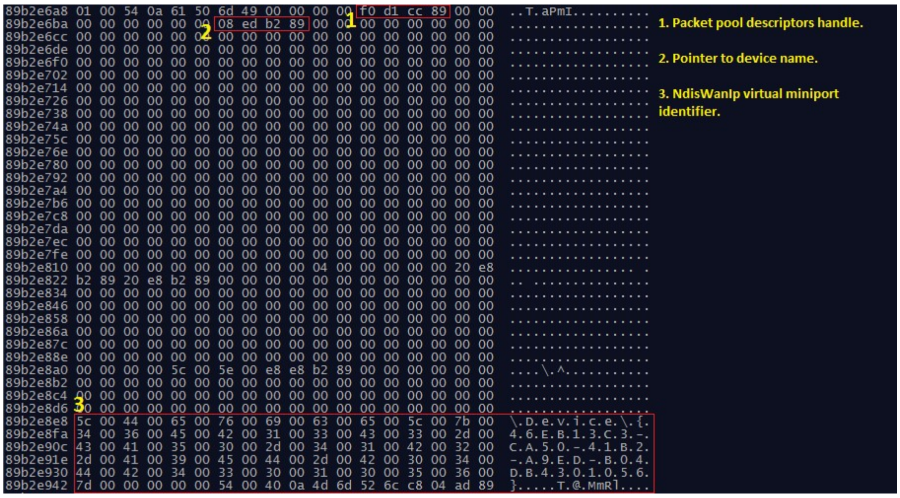


```
 89b2e458 size:   28 previous size:  270  (Allocated)  MmRl
 89b2e480 size:  140 previous size:   28  (Free)       MmIn
 89b2e5c0 size:   20 previous size:  140  (Allocated)  MmRl
 89b2e5e0 size:   a0 previous size:   20  (Free)       MmIn
 89b2e680 size:   20 previous size:   a0  (Allocated)  MmRl
 89b2e6a0 size:    8 previous size:   20  (Free)       Mmdl
*89b2e6a8 size:  2a0 previous size:    8  (Allocated) *aPmI
            Owning component : Unknown (update pooltag.txt)
 89b2e948 size:  200 previous size:  2a0  (Allocated)  MmRl
 89b2eb48 size:   78 previous size:  200  (Free )  Mmdl
 89b2ebc0 size:   60 previous size:   78  (Free)       MmIn
 89b2ec20 size:   40 previous size:   60  (Free )  Hal
```

Figure 24: aPmI tagged nonpaged pool buffer

Once the previously retrieved data (see figure 22) has been stored in the allocated buffer, a call to *NdisAllocatePacketPool* will take place. This used in order to allocate some memory to store packet descriptors. However, it also returns a handle to the allocated pool; in this case the pointer supplied to store that handle points inside the previously allocated "*aPmI*" tagged buffer. Then, *NdisAllocateBufferPool* is called to allocate some memory to store other buffer descriptors. Again, this will also return a handle to the previously allocated "*aPmI*" tagged buffer.

However, in some Windows versions a NULL returned handle value is valid. The call to *NdisAllocateMemory* that follows immediately after at *Epfwndis + 0x2BEA* is quite important. The base address of the allocation will also be stored in the familiar to us "*aPmI*" tagged buffer and it will point to the device name (see figure 25 on the following page, side note 2). The "*aPmI*" tagged buffer is going to be used as a context area to store per NIC device run-time state information for each of those that the intermediate driver under examination exposes a virtual miniport. At this stage we can examine the contents of the aforementioned tagged buffer.



```
89b2e6a8 01 00 54 0a 61 50 6d 49 00 00 00 00 f0 d1 cc 89 00 00  ..T.aPmI..........
89b2e6ba 00 00 00 00 00 00 08 ed b2 89 00 00 00 00 00 00 00 00  ..................
89b2e6cc 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e6de 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e6f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e702 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e714 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e726 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e738 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e74a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e75c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e76e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e780 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e792 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7a4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7b6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7c8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7da 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7ec 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7fe 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e810 00 00 00 00 00 00 00 00 00 00 04 00 00 00 00 00 20 e8  ................ .
89b2e822 b2 89 20 e8 b2 89 00 00 00 00 00 00 00 00 00 00 00 00  .. ...............
89b2e834 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e846 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e858 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e86a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e87c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e88e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e8a0 00 00 00 00 5c 00 5e 00 e8 e8 b2 89 00 00 00 00 00 00  ....\.^...........
89b2e8b2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e8c4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e8d6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e8e8 5c 00 44 00 65 00 76 00 69 00 63 00 65 00 5c 00 7b 00  \.D.e.v.i.c.e.\.{.
89b2e8fa 34 00 36 00 45 00 42 00 31 00 33 00 43 00 33 00 2d 00  4.6.E.B.1.3.C.3.-.
89b2e90c 43 00 41 00 35 00 30 00 2d 00 34 00 31 00 42 00 32 00  C.A.5.0.-.4.1.B.2.
89b2e91e 2d 00 41 00 39 00 45 00 44 00 2d 00 42 00 30 00 34 00  -.A.9.E.D.-.B.0.4.
89b2e930 44 00 42 00 34 00 33 00 30 00 31 00 30 00 35 00 36 00  D.B.4.3.0.1.0.5.6.
89b2e942 7d 00 00 00 00 00 54 00 40 0a 4d 6d 52 6c c8 04 ad 89  }.....T.@.MmRl....
```

1. Packet pool descriptors handle.

2. Pointer to device name.

3. NdisWanIp virtual miniport identifier.

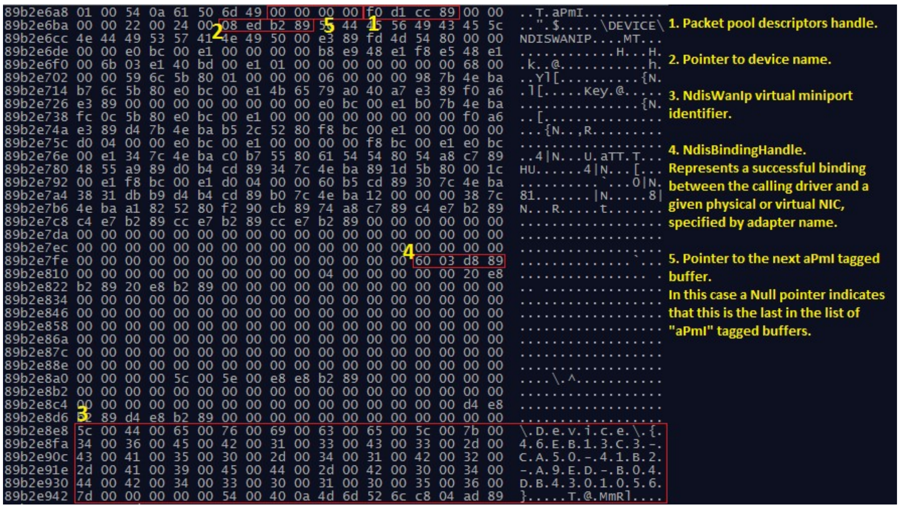
Figure 25: aPmI tagged buffer Stage1

We can also see the device name copied at this point to the allocated context area that will keep some per NIC device run-time state information (see figure 26).

```
kd> du 89b2ed08
89b2ed08  "\DEVICE\NDISWANIP"
```

Figure 26: Device name

Some more data is going to be copied back into the "*aPmI*" tagged buffer, and finally *NdisOpenAdapter* will be called in order to set up the binding between the protocol edge of *Epfwndis* and *NdisWanIp*. Our tagged buffer will now serve as a context area to maintain information about the state of the binding once it is established. Let's see what other information is now stored there.



```
89b2e6a8 01 00 54 0a 61 50 6d 49 00 00 00 00 f0 d1 cc 89 00 00  ..T.aPmI..........
89b2e6ba 00 00 22 00 24 00 08 ed b2 89 5c 44 45 56 49 43 45 5c  ..".$.....\DEVICE\
89b2e6cc 4e 44 49 53 57 41 4e 49 50 00 e3 89 fd 4d 54 80 00 00  NDISWANIP....MT...
89b2e6de 00 00 e0 bc 00 e1 00 00 00 00 b8 e9 48 e1 f8 e5 48 e1  ............H...H.
89b2e6f0 00 6b 03 e1 40 bd 00 e1 01 00 00 00 00 00 00 00 68 00  .k..@...........h.
89b2e702 00 00 59 6c 5b 80 01 00 00 00 06 00 00 00 98 7b 4e ba  ..Yl[..........{N.
89b2e714 b7 6c 5b 80 e0 bc 00 e1 4b 65 79 a0 40 a7 e3 89 f0 a6  .l[.....Key.@.....
89b2e726 e3 89 00 00 00 00 00 00 00 00 e0 bc 00 e1 b0 7b 4e ba  ...............{N.
89b2e738 fc 0c 5b 80 e0 bc 00 e1 00 00 00 00 00 00 00 00 f0 a6  ..[...............
89b2e74a e3 89 d4 7b 4e ba b5 2c 52 80 f8 bc 00 e1 00 00 00 00  ...{N..,R.........
89b2e75c d0 04 00 00 e0 bc 00 e1 00 00 00 00 f8 bc 00 e1 e0 bc  ..................
89b2e76e 00 e1 34 7c 4e ba c0 b7 55 80 61 54 54 80 54 a8 c7 89  ..4|N...U.aTT.T...
89b2e780 48 55 a9 89 d0 b4 cd 89 34 7c 4e ba 89 1d 5b 80 00 1c  HU......4|N...[...
89b2e792 00 e1 f8 bc 00 e1 d0 04 00 00 60 b5 cd 89 30 7c 4e ba  ..........`...0|N.
89b2e7a4 38 31 db b9 d4 b4 cd 89 b0 7c 4e ba 12 00 00 00 38 7c  81.......|N.....8|
89b2e7b6 4e ba a1 82 52 80 f2 90 cb 89 74 a8 c7 89 c4 e7 b2 89  N...R.....t.......
89b2e7c8 c4 e7 b2 89 cc e7 b2 89 cc e7 b2 89 00 00 00 00 00 00  ..................
89b2e7da 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7ec 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e7fe 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 03 d8 89  ..............`...
89b2e810 00 00 00 00 00 00 00 00 00 00 04 00 00 00 00 00 20 e8  ................ .
89b2e822 b2 89 20 e8 b2 89 00 00 00 00 00 00 00 00 00 00 00 00  .. ...............
89b2e834 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e846 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e858 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e86a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e87c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e88e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e8a0 00 00 00 00 5c 00 5e 00 e8 e8 b2 89 00 00 00 00 00 00  ....\.^...........
89b2e8b2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e8c4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 d4 e8  ..................
89b2e8d6 b2 89 d4 e8 b2 89 00 00 00 00 00 00 00 00 00 00 00 00  ..................
89b2e8e8 5c 00 44 00 65 00 76 00 69 00 63 00 65 00 5c 00 7b 00  \.D.e.v.i.c.e.\.{.
89b2e8fa 34 00 36 00 45 00 42 00 31 00 33 00 43 00 33 00 2d 00  4.6.E.B.1.3.C.3.-.
89b2e90c 43 00 41 00 35 00 30 00 2d 00 34 00 31 00 42 00 32 00  C.A.5.0.-.4.1.B.2.
89b2e91e 2d 00 41 00 39 00 45 00 44 00 2d 00 42 00 30 00 34 00  -.A.9.E.D.-.B.0.4.
89b2e930 44 00 42 00 34 00 33 00 30 00 31 00 30 00 35 00 36 00  D.B.4.3.0.1.0.5.6.
89b2e942 7d 00 00 00 00 00 54 00 40 0a 4d 6d 52 6c c8 04 ad 89  }.....T.@.MmRl....
```

1. Packet pool descriptors handle.

2. Pointer to device name.

3. NdisWanIp virtual miniport identifier.

4. NdisBindingHandle.
Represents a successful binding between the calling driver and a given physical or virtual NIC, specified by adapter name.

5. Pointer to the next aPmI tagged buffer.
In this case a Null pointer indicates that this is the last in the list of "aPmI" tagged buffers.

Figure 27: aPmI tagged buffer Stage2

At this stage, *Epfwndis* will trigger the process of initializing the virtual miniport by calling *NdisIMInitializeDeviceInstanceEx* (see figure 28 on the following page) from address *Epfwndis + 0x2DA0*.

```
NDIS_STATUS NdisIMInitializeDeviceInstanceEx(
  _In_      NDIS_HANDLE DriverHandle,
  _In_      PNDIS_STRING DriverInstance,
  _In_opt_  NDIS_HANDLE DeviceContext
);
```

Figure 28: NdisIMInitializeDeviceInstanceEx

We can see that this function accepts three parameters, which are:

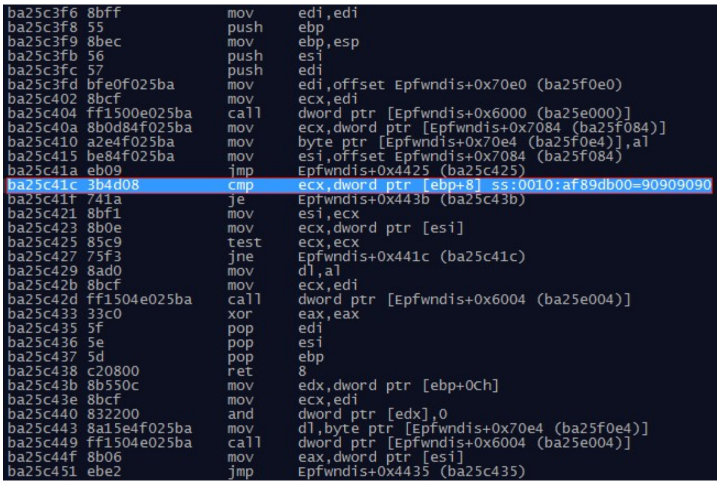
1. DriverHandle: handle returned by *NdisIMRegisterLayeredMiniport*
2. DriverInstance: Points to the *NdisWanIp* virtual miniport identifier ("\Device\46EB13C3-CA50-41B2-A9ED-B04DB4301056")
3. DeviceContext: Points to our "*aPmI*" allocated buffer which is used as a context area to keep information about a NIC device bound with the *Epfwndis* driver.

The number of "*aPmI*" allocated buffers depends on the amount of compatible adapters enabled in the host. If all adapters were disabled, we noticed that *EpFwndis* would only go through this process for *NdisWanIp*. Note that in figures 26 on the previous page and 27 on the preceding page, the pointer corresponding to side note two was only pointing to the same device name placed in another buffer. As we mentioned, the important information about the binding of any underlying miniport drivers is kept in the "*aPmI*" allocated buffers.

6.3 Triggering the vulnerability

When we started analyzing the *EpFwndis* driver by looking at the exposed *I/O Control Request Codes* (IOCTLs) [1] that can be used from userland in order to communicate with a kernel device driver using the *DeviceIoControl* function [2]. We managed to initially control the EIP by using *IOCTL 0x830020CC* and matching the necessary requirements regarding the contents of the input buffer.

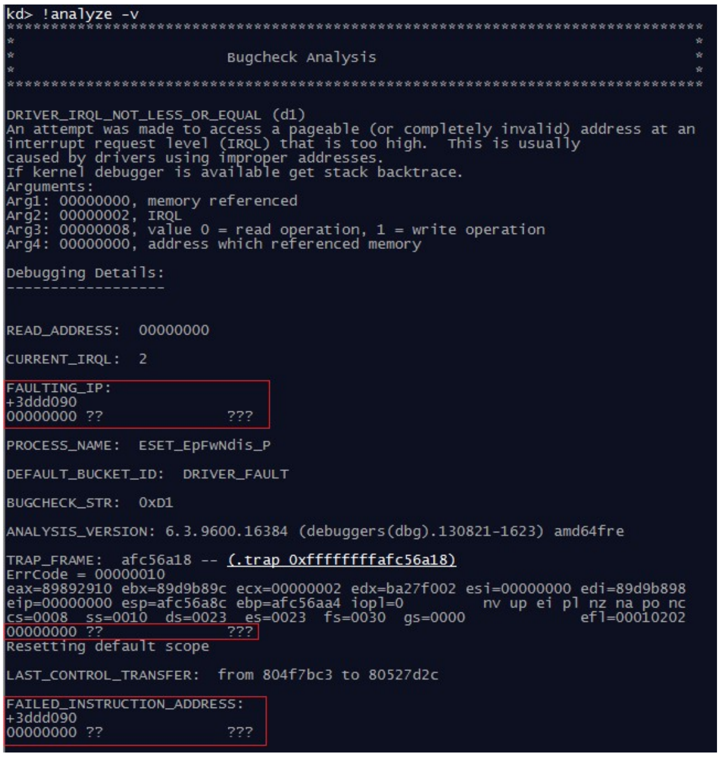
A subroutine located at address *Epfwndis + 0x43f6* (see figure 29 on the next page) is called when processing this specific IOCTL. Its purpose is to parse a list of "*aPmI*" tagged buffers allocated by *EpFwndis*. Each one of them serves as context area to keep information about a specific adapter that is bound to this ESET driver. The interesting part during this stage is that the driver will read a pointer from our input buffer which we control from userland and then will try to see if it matches any of the entries in the aforementioned list.



```
ba25c3f6 8bff            mov     edi,edi
ba25c3f8 55              push    ebp
ba25c3f9 8bec            mov     ebp,esp
ba25c3fb 56              push    esi
ba25c3fc 57              push    edi
ba25c3fd bfe0f025ba      mov     edi,offset Epfwndis+0x70e0 (ba25f0e0)
ba25c402 8bcf            mov     ecx,edi
ba25c404 ff1500e025ba    call    dword ptr [Epfwndis+0x6000 (ba25e000)]
ba25c40a 8b0d84f025ba    mov     ecx,dword ptr [Epfwndis+0x7084 (ba25f084)]
ba25c410 a2e4f025ba      mov     byte ptr [Epfwndis+0x70e4 (ba25f0e4)],al
ba25c415 be84f025ba      mov     esi,offset Epfwndis+0x7084 (ba25f084)
ba25c41a eb09            jmp     Epfwndis+0x4425 (ba25c425)
ba25c41c 3b4d08          cmp     ecx,dword ptr [ebp+8] ss:0010:af89db00=90909090
ba25c41f 741a            je      Epfwndis+0x443b (ba25c43b)
ba25c421 8bf1            mov     esi,ecx
ba25c423 8b0e            mov     ecx,dword ptr [esi]
ba25c425 85c9            test    ecx,ecx
ba25c427 75f3            jne     Epfwndis+0x441c (ba25c41c)
ba25c429 8ad0            mov     dl,al
ba25c42b 8bcf            mov     ecx,edi
ba25c42d ff1504e025ba    call    dword ptr [Epfwndis+0x6004 (ba25e004)]
ba25c433 33c0            xor     eax,eax
ba25c435 5f              pop     edi
ba25c436 5e              pop     esi
ba25c437 5d              pop     ebp
ba25c438 c20800          ret     8
ba25c43b 8b550c          mov     edx,dword ptr [ebp+0Ch]
ba25c43e 8bcf            mov     ecx,edi
ba25c440 832200          and     dword ptr [edx],0
ba25c443 8a15e4f025ba    mov     dl,byte ptr [Epfwndis+0x70e4 (ba25f0e4)]
ba25c449 ff1504e025ba    call    dword ptr [Epfwndis+0x6004 (ba25e004)]
ba25c44f 8b06            mov     eax,dword ptr [esi]
ba25c451 ebe2            jmp     Epfwndis+0x4435 (ba25c435)
```

Figure 29: Parsing aPmI tagged buffers list

Figure 29 includes a dummy pointer (0x90909090) which of course wouldn't match any of the allocated "*aPmI*" tagged buffers, but it clearly shows that we can control this parameter from userland. If a match is found, then on completion of this request the protocol function registered as `RECEIVE_COMPLETE _HANDLER` in the corresponding `NDIS_PROTOCOL_CHARACTERISTICS` structure for that driver will be called. In our case, if the pointer matches the "aPmI" tagged buffer that keeps information about the *NdisWanIp*, then *wanarp!WanNdisReceiveComplete* is going to be called with the following results.

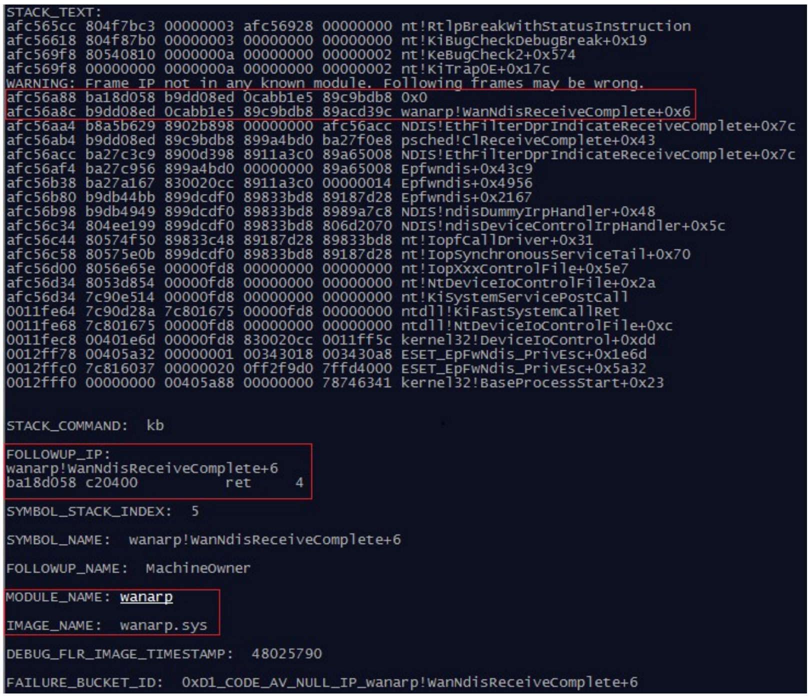


```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: 00000000, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000008, value 0 = read operation, 1 = write operation
Arg4: 00000000, address which referenced memory

Debugging Details:
------------------


READ_ADDRESS:  00000000

CURRENT_IRQL:  2

FAULTING_IP:
+3ddd090
00000000 ??              ???

PROCESS_NAME:  ESET_EpFwNdis_P

DEFAULT_BUCKET_ID:  DRIVER_FAULT

BUGCHECK_STR:  0xD1

ANALYSIS_VERSION: 6.3.9600.16384 (debuggers(dbg).130821-1623) amd64fre

TRAP_FRAME:  afc56a18 -- (.trap 0xffffffffafc56a18)
ErrCode = 00000010
eax=89892910 ebx=89d9b89c ecx=00000002 edx=ba27f002 esi=00000000 edi=89d9b898
eip=00000000 esp=afc56a8c ebp=afc56aa4 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202
00000000 ??              ???
Resetting default scope

LAST_CONTROL_TRANSFER:  from 804f7bc3 to 80527d2c

FAILED_INSTRUCTION_ADDRESS:
+3ddd090
00000000 ??              ???
```

Figure 30: EIP == NULL

The execution flow was transferred at address *0x00000000* which was not allocated and this caused our system to crash.

Let's see what other information Windbg [5] can provide to us.



```
STACK_TEXT:
afc565cc 804f7bc3 00000003 afc56928 00000000 nt!RtlpBreakWithStatusInstruction
afc56618 804f87b0 00000003 00000000 00000000 nt!KiBugCheckDebugBreak+0x19
afc569f8 80540810 0000000a 00000000 00000002 nt!KeBugCheck2+0x574
afc569f8 00000000 0000000a 00000000 00000002 nt!KiTrap0E+0x17c
WARNING: Frame IP not in any known module. Following frames may be wrong.
afc56a88 ba18d058 b9dd08ed 0cabb1e5 89c9bdb8 0x0
afc56a8c b9dd08ed 0cabb1e5 89c9bdb8 89acd39c wanarp!WanNdisReceiveComplete+0x6
afc56aa4 b8a5b629 8902b898 00000000 afc56acc NDIS!EthFilterDprIndicateReceiveComplete+0x7c
afc56ab4 b9dd08ed 89c9bdb8 899a4bd0 ba27f0e8 psched!ClReceiveComplete+0x43
afc56acc ba27c3c9 8900d398 8911a3c0 89a65008 NDIS!EthFilterDprIndicateReceiveComplete+0x7c
afc56af4 ba27c956 899a4bd0 00000000 89a65008 Epfwndis+0x43c9
afc56b38 ba27a167 830020cc 8911a3c0 00000014 Epfwndis+0x4956
afc56b80 b9db44bb 899dcdf0 89833bd8 89187d28 Epfwndis+0x2167
afc56b98 b9db4949 899dcdf0 89833bd8 8989a7c8 NDIS!ndisDummyIrpHandler+0x48
afc56c34 804ee199 899dcdf0 89833bd8 806d2070 NDIS!ndisDeviceControlIrpHandler+0x5c
afc56c44 80574f50 89833c48 89187d28 89833bd8 nt!IopfCallDriver+0x31
afc56c58 80575e0b 899dcdf0 89833bd8 89187d28 nt!IopSynchronousServiceTail+0x70
afc56d00 8056e65e 00000fd8 00000000 00000000 nt!IopXxxControlFile+0x5e7
afc56d34 8053d854 00000fd8 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
afc56d34 7c90e514 00000fd8 00000000 00000000 nt!KiSystemServicePostCall
0011fe64 7c90d28a 7c801675 00000fd8 00000000 ntdll!KiFastSystemCallRet
0011fe68 7c801675 00000fd8 00000000 00000000 ntdll!NtDeviceIoControlFile+0xc
0011fec8 00401e6d 00000fd8 830020cc 0011ff5c kernel32!DeviceIoControl+0xdd
0012ff78 00405a32 00000001 00343018 003430a8 ESET_EpFwNdis_PrivEsc+0x1e6d
0012ffc0 7c816037 00000020 0ff2f9d0 7ffd4000 ESET_EpFwNdis_PrivEsc+0x5a32
0012fff0 00000000 00405a88 00000000 78746341 kernel32!BaseProcessStart+0x23


STACK_COMMAND:  kb

FOLLOWUP_IP:
wanarp!WanNdisReceiveComplete+6
ba18d058 c20400          ret     4

SYMBOL_STACK_INDEX:  5

SYMBOL_NAME:  wanarp!WanNdisReceiveComplete+6

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: wanarp

IMAGE_NAME:  wanarp.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  48025790

FAILURE_BUCKET_ID:  0xD1_CODE_AV_NULL_IP_wanarp!WanNdisReceiveComplete+6
```

Figure 31: Call stack trace

In figure 29 on the previous page we notice that the execution flow was transferred on the NULL page after calling *wanarp!WanNdisReceiveComplete*. The return address was expected to be *wanarp!WanNdisReceiveComplete+6*, but of course we never arrived there since things went wrong once the call was performed. Since the root cause of this issue is already explained in section 5 on page 8, it is probably the best time for the reader to go back and have a quick look at the information provided in that section and more specifically in figures 13 on page 9 and 14 on page 9.

6.4 Leaking NdisWanIp Device Context Kernel Pointer

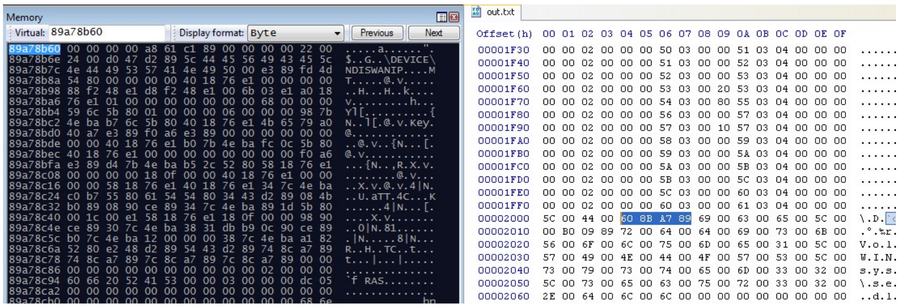
As we have seen so far, in order to trigger this vulnerability it is necessary to match the pointer to the device context area regarding *NdisWanIp* device. Since this is a value that we can control from userland, an attacker could attempt to bruteforce it. However, this would probably take some time to achieve and it could also have some impact in the stability of the host. During our tests, both of the aforementioned situations occurred during bruteforcing attempts.

Fortunately, there is a much better way to exploit this vulnerability without having to actually bruteforce this magic pointer value. In fact, we can either directly retrieve this value or do a very small amount of attempts over a some data leaked from the kernel address space. By using IOCTL *0x830020C4* we were able to leak this pointer from kernel back to userland and fit it nicely in the input buffer for the next IOCTL that we discussed about in the previous section. In reality, this IOCTL can be used to retrieve data from those "*aPmI*" tagged buffers. The good thing, for us, is that the data returned can be up to a size of *0x2204* bytes which means that if we declare a big enough output buffer in the call to the *DeviceIoControl* [2] function then we might be able to leak extra data from kernel address space.

In fact, the data returned in that buffer from kernel space will include the valid pointer to the "*aPmI*" buffer that holds information about the *NdisWanIp* device. In a few words, the handler for this IOCTL will basically go through the list of the context areas allocated for each device bound to the intermediate driver under examination and will return this data back to userland. Generally, a host wouldn't have more than 1 or 2 extra adapters bound to the intermediate driver, apart from the *NdisWanIp* which is always initialized, so the extra memory leaked contains that useful information. With no adapters enabled the magic pointer was located at offset *0x2004* (see figure 32) in the kernel leaked memory buffer, while with 1 adapter enabled the same pointer was located at offset *0x2008* in the output buffer.

However, in this particular case leaking the pointer from kernel was not necessary. These pointers to the "*aPmI*" buffers are stored in a buffer inside the loaded *EpFwndis* kernel module and the proprietary handler for the IOCTL *0x830020CC* allows also to specify from where it will read the input pointer. So in practice we could enumerate for the loaded drivers, get the image base of the *EpFwndis* driver, add the RVA of that buffer and send this in the input buffer as pointer from where the magic value will be read. This of course, makes the attack module-version and build specific. Instead, by leaking the pointer using the method we previously described is much more universal since we don't need to know a specific RVA. This means that we can use it to attack also other vulnerable NDIS 5.x intermediate drivers that might directly accept the magic pointer from the input buffer instead of also accepting an address from where to read this pointer.

So, in this case since we control both, we leak the magic pointer from kernel and we instruct the driver to read it from the input buffer that we send through the call to the *DeviceIoControl API*. Figure 32 demonstrates the leaked pointer from kernel.



```
89a78b60 00 00 00 00 a8 61 c1 89 00 00 00 00 22 00  .....a......".
89a78b6e 24 00 d0 47 d2 89 5c 44 45 56 49 43 45 5c  $..G..\DEVICE\
89a78b7c 4e 44 49 53 57 41 4e 49 50 00 e3 89 fd 4d  NDISWANIP....M
89a78b8a 54 80 00 00 00 00 40 18 76 e1 00 00 00 00  T.....@.v.....
89a78b98 88 f2 48 e1 d8 f2 48 e1 00 6b 03 e1 a0 18  ..H...H..k....
89a78ba6 76 e1 01 00 00 00 00 00 00 00 68 00 00 00  v.........h...
89a78bb4 59 6c 5b 80 01 00 00 00 06 00 00 00 98 7b  Yl[..........{
89a78bc2 4e ba b7 6c 5b 80 40 18 76 e1 4b 65 79 a0  N..l[.@.v.Key.
89a78bd0 40 a7 e3 89 f0 a6 e3 89 00 00 00 00 00 00  @.............
89a78bde 00 00 40 18 76 e1 b0 7b 4e ba fc 0c 5b 80  ..@.v..{N...[.
89a78bec 40 18 76 e1 00 00 00 00 00 00 00 00 f0 a6  @.v...........
89a78bfa e3 89 d4 7b 4e ba b5 2c 52 80 58 18 76 e1  ...{N..,R.X.v.
89a78c08 00 00 00 00 18 0f 00 00 40 18 76 e1 00 00  ........@.v...
89a78c16 00 00 58 18 76 e1 40 18 76 e1 34 7c 4e ba  ..X.v.@.v.4|N.
89a78c24 c0 b7 55 80 61 54 54 80 34 43 d2 89 08 4b  ..U.aTT.4C...K
89a78c32 b0 89 08 90 ce 89 34 7c 4e ba 89 1d 5b 80  ......4|N...[.
89a78c40 00 1c 00 e1 58 18 76 e1 18 0f 00 00 98 90  ....X.v.......
89a78c4e ce 89 30 7c 4e ba 38 31 db b9 0c 90 ce 89  ..0|N.81......
89a78c5c b0 7c 4e ba 12 00 00 00 38 7c 4e ba a1 82  .|N.....8|N...
89a78c6a 52 80 e2 48 d2 89 54 43 d2 89 74 8c a7 89  R..H..TC..t...
89a78c78 74 8c a7 89 7c 8c a7 89 7c 8c a7 89 00 00  t...|...|.....
89a78c86 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ..............
89a78c94 60 66 20 52 41 53 00 00 03 00 00 00 dc 05  `f RAS........
89a78ca2 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
```

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00001F30  00 00 02 00 00 00 50 03 00 00 51 03 04 00 00 00  ......
00001F40  00 00 02 00 00 00 51 03 00 00 52 03 04 00 00 00  ......
00001F50  00 00 02 00 00 00 52 03 00 00 53 03 04 00 00 00  ......
00001F60  00 00 02 00 00 00 53 03 00 20 53 03 04 00 00 00  ......
00001F70  00 00 02 00 00 00 54 03 00 80 55 03 04 00 00 00  ......
00001F80  00 00 02 00 00 00 56 03 00 00 57 03 04 00 00 00  ......
00001F90  00 00 02 00 00 00 57 03 00 10 57 03 04 00 00 00  ......
00001FA0  00 00 02 00 00 00 58 03 00 00 59 03 04 00 00 00  ......
00001FB0  00 00 02 00 00 00 59 03 00 00 5A 03 04 00 00 00  ......
00001FC0  00 00 02 00 00 00 5A 03 00 00 5B 03 04 00 00 00  ......
00001FD0  00 00 02 00 00 00 5B 03 00 00 5C 03 04 00 00 00  ......
00001FE0  00 00 02 00 00 00 5C 03 00 00 60 03 04 00 00 00  ......
00001FF0  00 00 02 00 00 00 60 03 00 00 61 03 04 00 00 00  ......
00002000  5C 00 44 00 60 8B A7 89 69 00 63 00 65 00 5C 00  \.D.
00002010  00 B0 09 89 72 00 64 00 64 00 69 00 73 00 6B 00  .°.‰r.
00002020  56 00 6F 00 6C 00 75 00 6D 00 65 00 31 00 5C 00  V.o.l.
00002030  57 00 49 00 4E 00 44 00 4F 00 57 00 53 00 5C 00  W.I.N.
00002040  73 00 79 00 73 00 74 00 65 00 6D 00 33 00 32 00  s.y.s.
00002050  5C 00 73 00 65 00 63 00 75 00 72 00 33 00 32 00  \.s.e.
00002060  2E 00 64 00 6C 00 6C 00 00 00 00 00 00 00 00 00  ..d.l.
```

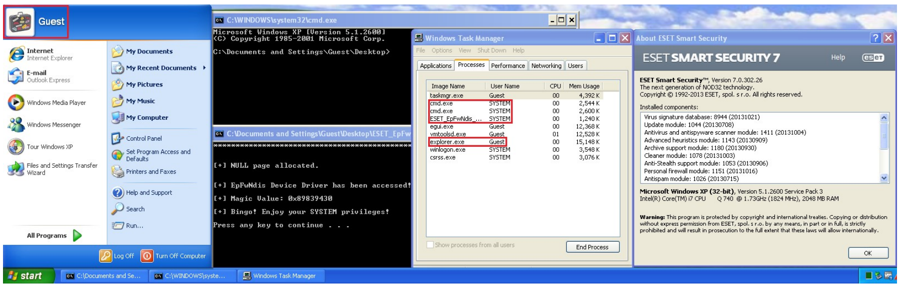
Figure 32: Leaked pointer from kernel

6.5 Privilege Escalation

At this point, we have all the necessary pieces of the puzzle in place and it's time for us to enjoy the view. So, just to put everything together these are the steps used for exploiting a vulnerable NDIS 5.x intermediate driver in Windows XP and Windows Server 2003:

1. Allocate NULL page.
2. Place a trampoline to our payload.
3. Leak kernel pointer to *NdisWanIp* device context area using IOCTL *0x830020C4*.
4. Trigger a call to *wanarp!WanNdisReceiveComplete* using IOCTL *0x830020CC*.
5. Execute payload.

As you can notice in the figure above, Windows explorer is running as 'Guest' while the processes related to our exploit are now running as 'SYSTEM'. This is



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Guest\Desktop>
```

```
****************************************************

[+] NULL page allocated.

[+] EpFwNdis Device Driver has been accessed!
[+] Magic Value: 0x89839430
[+] Bingo! Enjoy your SYSTEM privileges!
Press any key to continue . . .
```

Image Name	User Name	CPU	Mem Usage
taskmgr.exe	Guest	00	4,392 K
cmd.exe	SYSTEM	00	2,544 K
cmd.exe	SYSTEM	00	2,600 K
ESET_EpFwNdis_...	SYSTEM	00	1,240 K
egui.exe	Guest	00	12,368 K
vmtoolsd.exe	Guest	01	12,528 K
explorer.exe	Guest	00	15,148 K
winlogon.exe	SYSTEM	00	3,548 K
csrss.exe	SYSTEM	00	3,076 K

ESET Smart Security™, Version 7.0.302.26
The next generation of NOD32 technology.
Copyright © 1992-2013 ESET, spol. s r.o. All rights reserved.

Installed components:
Virus signature database: 8944 (20131021)
Update module: 1044 (20130708)
Antivirus and antispyware scanner module: 1411 (20131004)
Advanced heuristics module: 1143 (20130909)
Archive support module: 1180 (20130930)
Cleaner module: 1078 (20131003)
Anti-Stealth support module: 1053 (20130906)
Personal firewall module: 1151 (20131016)
Antispam module: 1026 (20130715)

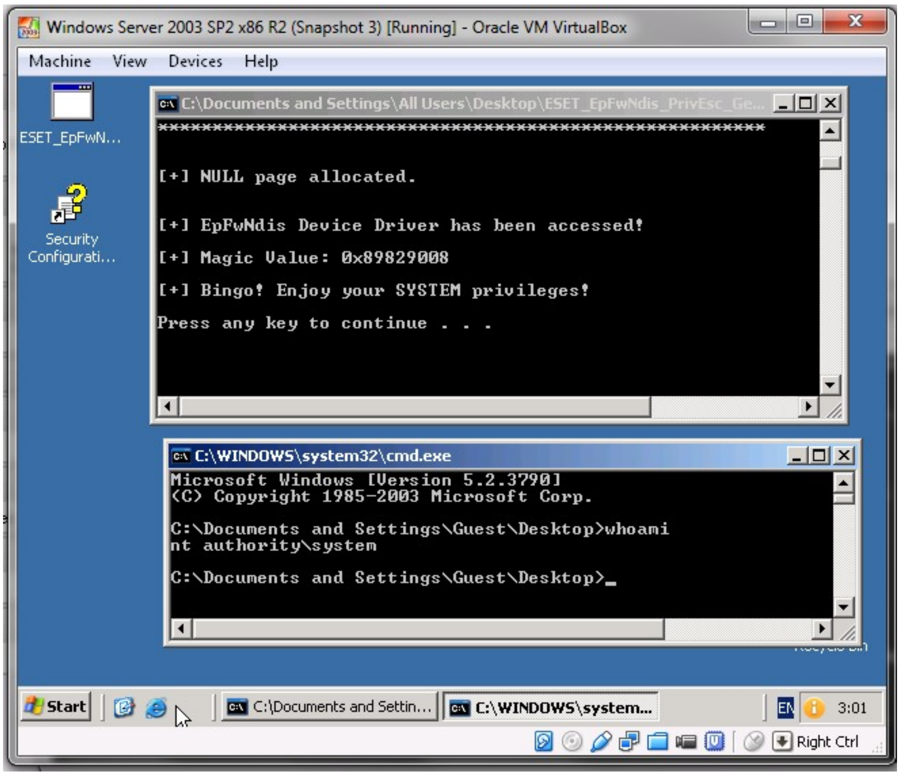
Microsoft Windows XP (32-bit), Version 5.1.2600 Service Pack 3
Intel(R) Core(TM) i7 CPU Q 740 @ 1.73GHz (1824 MHz), 2048 MB RAM

Warning: This program is protected by copyright and international treaties. Copying or distribution without express permission from ESET, spol. s r.o. by any means, in part or in full, is strictly prohibited and will result in prosecution to the full extent that these laws will allow internationally.

Figure 33: Privilege Escalation in Windows XP SP3

because we used an XP specific payload to parse the EPROCESS structures of all active processes in search for the SYSTEM process. We know that this process has PID 4, so once found we steal the pointer to its security access token and then we substitute the token pointer of the parent exploit process which initially runs as ’Guest’ with that one. The rest is history since the child processes of our exploit will also inherit the same token which now is the one belonging to the SYSTEM process.

The following screenshot is taken from a Windows Server 2003 R2 virtual machine, where we exploited the vulnerability through the *ESET Endpoint Security* product.



```
**************************************************************

[+] NULL page allocated.

[+] EpFwNdis Device Driver has been accessed!
[+] Magic Value: 0x89829008
[+] Bingo! Enjoy your SYSTEM privileges!
Press any key to continue . . .
```

```
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\Documents and Settings\Guest\Desktop>whoami
nt authority\system

C:\Documents and Settings\Guest\Desktop>_
```

Figure 34: Privilege Escalation in Windows Server 2003 R2

Notice that in the parent exploit process we also output the leaked pointer to the context area allocated by *EpFwndis* for the *NdisWanIp* device.

7 VENDORS AFFECTED

Multiple products of the following vendors that are built for Windows XP and Windows Server 2003 (R2 included) were affected by this issue and almost certainly many other products from other vendors are currently vulnerable. Vendors still supporting products originally built for these Windows operating systems should revise immediately their code and make sure that their NDIS 5.x intermediate drivers are not affected. In general NDIS 5.x intermediate drivers that expose the aforementioned IOCTL codes are likely to be vulnerable to privilege escalation. The following vendors (except from the last one in the list) have successfully patched their drivers, thus they are not affected anymore of this issue.

1. ESET: CVE-2014-4973
2. G Data: CVE-2014-9332
3. K7 Computing: CVE-2015-3444
4. QuickHeal/Seqrite: CVE-2015-3899

8 CONCLUSION

Vulnerabilities caused by design errors are definitely the most interesting to discover and exploit. In this white paper we went through a series of things that when put together they can be used by a malicious attacker to leverage his privileges and completely compromise the affected host.

Since Windows Server 2003 is also affected, this can be of great importance since compromising one host can potentially lead to compromise an entire or part of a corporate network infrastructure. This was proved to be a great lesson regarding the levels of difficulty in applying computer systems security. In other words, this is how two completely independent design errors from different vendors can generate an unpredictable, but exploitable situation with a highly severe impact in the context of computer security.

9 ACKNOWLEDGEMENTS

I would like to thank Portcullis Computer Security Ltd (`https://www.portcullis-security.com`) for giving me the time to write part of this white paper during my employment with them. This white paper was completed and published when I was not employed by Portcullis Computer Security Ltd anymore, hence the company assumes no responsibility for its contents.

Furthermore, I would like to thank from the bottom of my heart my friends Francisco Ribeiro and Matthieu Bonetti for their useful suggestions and technical reviews during the final stage of preparation for publishing this white paper. Last but not least, this paper is dedicated to all my friends. They know who they are.

REFERENCES

[1] Microsoft Dev Center. Device Input and Output Control (IOCTL). https://msdn.microsoft.com/en-us/library/windows/desktop/aa363219 Accessed: 2015-08-04.

[2] Microsoft Dev Center. DeviceIoControl function. https://msdn.microsoft.com/en-us/library/windows/desktop/aa363216 Accessed: 2015-08-04.

[3] Microsoft Dev Center. Introduction to Intermediate Drivers (NDIS 5.1). https://msdn.microsoft.com/en-us/library/windows/hardware/ff548970 Accessed: 2015-08-04.

[4] Microsoft Dev Center. NDIS Filter Intermediate Drivers (NDIS 5.1). https://msdn.microsoft.com/en-us/library/windows/hardware/ff556949 Accessed: 2015-08-04.

[5] Microsoft Hardware Dev Center. Debugging Tools for Windows (WinDbg, KD, CDB, NTSD). https://msdn.microsoft.com/en-us/library/windows/hardware/ff551063 Accessed: 2016-08-01.

[6] Microsoft Hardware Dev Center. NDIS Protocol Drivers. https://msdn.microsoft.com/en-us/library/windows/hardware/ff566823 Accessed: 2015-08-04.

[7] Microsoft Hardware Dev Center. Obsolete NDISXxx Functions and Macros for Windows 2000 and Later, and for Windows Me and Later. https://msdn.microsoft.com/en-us/library/windows/hardware/ff559181 Accessed: 2015-08-04.

[8] Microsoft Hardware Dev Center. Registering an Intermediate Driver as a Miniport (NDIS 5.1). https://msdn.microsoft.com/en-us/library/windows/hardware/ff563305 Accessed: 2015-08-04.

[9] Microsoft Hardware Dev Center. Registering as an NDIS Protocol Driver (NDIS 5.1). https://msdn.microsoft.com/en-us/library/windows/hardware/ff563310 Accessed: 2015-08-04.

[10] Microsoft Developer Network. NDIS Miniport Drivers. https://msdn.microsoft.com/en-us/en Accessed: 2015-08-04.

[11] Microsoft TechNet. Network Driver Interface Specification. https://technet.microsoft.com/en-gb/library/cc958797.aspx. Accessed: 2015-08-04.