Detectando hooks en procesos desde un módulo de Kernel

Por Hendrix

Hendrix@elhacker.net

En el presente artículo se va a tratar la detección de API Hooking en procesos mediante un módulo de Kernel. Un breve índice sobre los puntos que voy a tratar:

1. Diferencia entre manejo de direcciones en modo Kernel y modo Usuario
2. Trabajando sobre el proceso en modo Kernel
3. Leyendo memoria y código final

1. Diferencia entre manejo de direcciones en modo Kernel y modo Usiario:

Como ya saben, no es lo mismo trabajar con memorias en un módulo de Kernel que en un proceso corriendo en modo usuario. La principal diferencia es que un módulo Kernel trabaja con direcciones físicas y los procesos con memoria virtual. Vamos a ver mas detalladamente que es cada cosa.

- Memoria Virtual: La memoria virtual es la memoria que otorga el SistemaOperativo a un proceso para que este pueda trabajar. Se le llama virtual, ya que las direcciones de memoria son asignadas por el SO, con esto, entre muchas otras cosas, se bloquea el acceso en memoria a otros procesos, ya que la memoria que puede alcanzar el proceso solamente pertenece a el mismo (ningún otro proceso es cargado en el espacio de memoria virtual). Muchas veces, al depurar varios programas a la vez, se puede observar que comparten direcciones de memoria (por ejemplo, la dirección 0x0040C8B3) sin embargo, contienen diferentes instrucciones. Esto es debido a la memoria virtual, cada proceso esta encapsulado para que trabaja con sus propias direcciones sin interferir en la ejecución de los demás.
- Memoria física: La memoria física, en terminos de Sistemas Operativos y su kernel, es aquella en la cual reside el Kernel del SistemaOperativo una vez el PC es cargado. A diferencia de la Memoria Virtual, el codigo ejecutado en la memoria física no esta encapsulado, lo que permite que los módulos lean/escriban sobre la memoria que no les corresponde (aunque para ello, se ideó el Registro de Control en los procesadores i386 que, entre otras cosas, protege ciertas páginas de memoria, aunque bien es sabido que esta medida de protección se puede saltar con unas cuantas lineas en ensamblador).

Al querer trabajar con procesos desde un driver, se nos presenta la siguiente pregunta: ¿Como trabajamos con memoria virtual desde la memoria física? En el siguiente punto responderemos a esta pregunta.

2. Trabajando sobre el proceso en Modo kernel

Para manipular la memoria virtual de los procesos, existen varias API's que nos permiten trabajar con ellos, nosotros nos centraremos concretamente en dos de ellas, *ObReferenceObjectByHandle* y *KeStackAttachProcess*.

Los que hacen estas dos API's, a grandes rasgos, es dejarnos trabajar dentro del espacio de memoria virtual del proceso, una vez terminemos de trabajar, tenemos que salir de dicho espacio, para ello usaremos las API's *KeUnstackDetachProcess* y *ObfDereferenceObject*.

Este método nos permite tocar lo que queramos dentro del proceso y, aunque sea más peligroso (si hay algún error, nos tira BSOD), podemos tocar lo que queramos sobre cualquier proceso.

Una vez sabemos las API's a usar y lo que nos permiten hacer, pasemos a la parte importante.

3. Leyendo memoria y código final

Lo que e echo yo para mi módulo de Kernel, es crearme una función, la cual se le pasa por parámetro el Handle del proceso con el que trabajaremos, la dirección donde leer y el valor con el que se va a comparar y nos dice si existe hook o no.

En mi aplicación, e "harcodeado" los valores del inicio de la API, ya que el tema principal era el de mirar si hay hook o no, lo ideal sería que desde una aplicación en modo usuario, coger estos valores (cargar la Dll, ir a la dirección donde esta la API y coger los valores requeridos). Para evitar problemas, la longitud del buffer es de 1 int (por ejemplo, un patrón válido sería: 0x8AAAAAAA).

El siguiente paso es obtener el patrón (los primeros bytes de la API original) de la API a verificar, en el ejemplo, hookeare la API MessageBoxExA. Para ello, abrimos el WinDbg, abrimos un ejecutable cualquiera y escribimos lo siguiente:

u MessageBoxA

Lo que nos da el siguiente resultado:

```
USER32!MessageBoxExA:
7e3d085c 8bff            mov     edi,edi
7e3d085e 55              push    ebp
7e3d085f 8bec            mov     ebp,esp
7e3d0861 6aff            push    0FFFFFFFFh
7e3d0863 ff7518          push    dword ptr [ebp+18h]
7e3d0866 ff7514          push    dword ptr [ebp+14h]
7e3d0869 ff7510          push    dword ptr [ebp+10h]
7e3d086c ff750c          push    dword ptr [ebp+0Ch]
```

La primera columna es la dirección en donde se carga, la segunda, lo que nos interesa, los bytes de la API. Y como estamos en una arquitectura Little Endian, el patrón de bytes es el siguiente: 0x8B55FF8B.

Ahora que ya tenemos la dirección de la API y el patrón, podemos empezar a codear. Lo que tenemos que hacer es lo siguiente:

1. Entrar en el espacio del proceso con el que trabajar
2. Copiar los primeros bytes dentro de un buffer
3. Salir del espacio del proceso
4. Comparar para ver si hay hook

Para el primer paso, nos valdremos de las Apis antes comentadas, pero antes, tenemos que obtener el Handle del proceso. La función que e programado para ello es la siguiente:

```
HANDLE GetHandlByPid(int pid)
{
	HANDLE phandle;
  OBJECT_ATTRIBUTES oa;
  CLIENT_ID cid;
	NTSTATUS ret = 0;

	__try
	{
		InitializeObjectAttributes(&oa,NULL,0,NULL,NULL);
		cid.UniqueProcess=(HANDLE)pid;
		cid.UniqueThread=NULL;

		ret=ZwOpenProcess(&phandle,GENERIC_READ,&oa,&cid);

		if(!NT_SUCCESS(ret))
		{
			return 0;
		}
	}
```

```
	__except(EXCEPTION_EXECUTE_HANDLER)
	{
		DbgPrint("Error en el procesamiento de GetHandlByPid");
	}

	return phandle;
}
```

Una vez hemos obtenido el Handle, la siguiente función que e programado es la que propiamente verifica si hay Hook o no. Pego el código y luego lo explico.

```
NTSTATUS VerificaHook(HANDLE ProcHandle, PVOID Direccion, int
CorrectBytes)
{
 PVOID ProcObj=NULL;
 KAPC_STATE ApcState;
 POBJECT_TYPE PsProcType = 0;
 int i=0;
 int Buff;
 int Hook = 0;
 __try
 {
	ObReferenceObjectByHandle(ProcHandle, 0x10, PsProcType,
KernelMode, &ProcObj, NULL);
	KeStackAttachProcess(ProcObj, &ApcState);
	RtlCopyMemory(&Buff, Direccion, sizeof(int));
	KeUnstackDetachProcess(&ApcState);
	ObfDereferenceObject(ProcObj);

	if(Buff == CorrectBytes)
	{
		//DbgPrint("No hay hook");
		Hook = 0;
	}
	else
	{
		//DbgPrint("Hay Hook");
		DbgPrint("Buff: %x",Buff);
		DbgPrint("CrBt: %x", CorrectBytes);
		Hook = 1;
	}

 }

 __except(EXCEPTION_EXECUTE_HANDLER)
 {
	DbgPrint("Error en el procesamiento de VerificaHook");
```

```
        return -1;
 }
 return Hook;
}
```

Lo primero que hacemos es referenciar el Handle, una vez echo esto, nos metemos en el espacio de memoria del proceso con *KeStackAttachProcess* y una vez estamos dentro del espacio de memoria virtual del proceso, lo único que tengo que hacer es leer la memoria y guardarlo en un Buffer y luego comparar este con los bytes originales que antes habíamos obtenido.

Si quisiéramos reparar el hook, con usar la API *RtlMoveMemory* podríamos sobrescribir la parte hookeada con sus bytes originales.

El cuerpo del módulo restante es el siguiente:

```
#include <ntddk.h>
#include "ntifs.h"
#include <string.h>
#include <stdio.h>

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


NTSTATUS DriverEntry( PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
        HANDLE myHand;
        int Ret = 0;
        PVOID Dir = (PVOID)0x7E3D0774; //Direccion de MessageBoxExA
        int BitesOriginales = 0x8B55FF8B; //Bytes originales de la API
   DriverObject->DriverUnload=Salir;

        DbgPrint("Cargado");
        myHand = GetHandlByPid(2016); //Obtenemos el Handle
        if(myHand == 0)
        {
                DbgPrint("Error al sacar el Handle");
                return STATUS_SUCCESS;
        }
        else
        {
                DbgPrint("Handle: 0x%x",myHand);
```

```
	}

	Ret = (int)VerificaHook(myHand,Dir,BitesOriginales); //Verificamos
Hooks

	switch(Ret)
	{
		case 0:
			DbgPrint("No hay hook");
			break;
		case 1:
			DbgPrint("Hay hook");
			break;
		case -1:
			DbgPrint("Error en la función");
			break;
	}
	ZwClose(myHand);
	return  STATUS_SUCCESS;
}
```

Con esto ya tenemos una pequeña base para programar un programa sencillo para detectar hooks en modo usuario y poderlos reparar desde Kernel, aunque el objetivo de este artículo es el de trabajar en el espacio de procesos desde un módulo de Kernel.

Un Saludo

Hendrix.