

# SSDT Y COMO RESOLVEMOS LOS SYSCALLS.

Como estoy trabajando bastante con syscalls, quería dejar escrito como resolverlos para saber a qué función en kernel terminan saltando, desde una función en user.

De paso es divertido trastear un poco, y siempre se aprende.

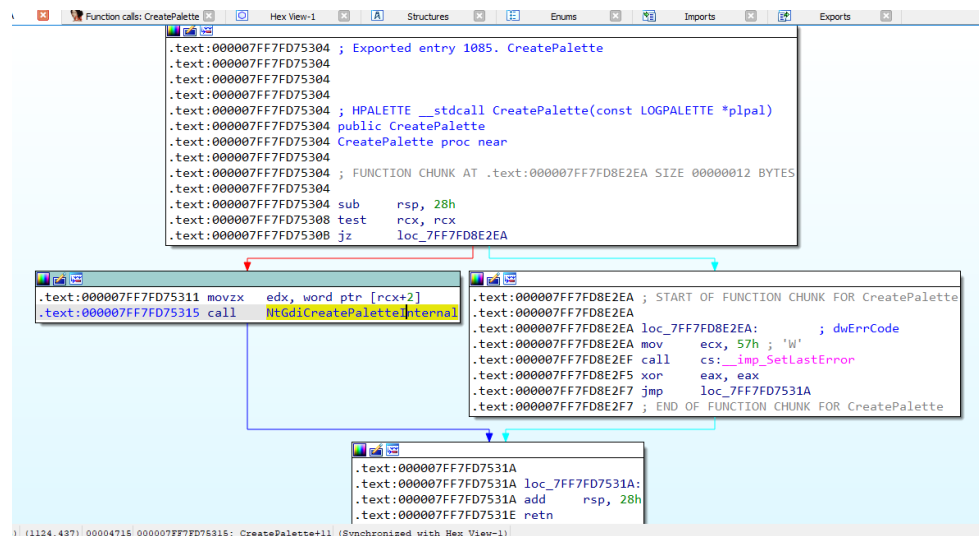
Se que al leer esto muchos dirán bueno, pero pones un breakpoint allí en el syscall y traceas y listo (no es tan alegre el tema ya veremos jeje), igual es bueno conocer cómo trabaja la SSDT y lo que agrega conocimiento no daña.

Vayamos mirando algunos conceptos.

## QUE ES LA SSDT?

Cuando se realiza una llamada al sistema, se utiliza un índice para acceder a la función concreta que se desea utilizar, dentro de una tabla, la cual se conoce como tabla SSDT.

Por ejemplo, voy a tomar una función cualquiera de user de las tantas que acceden al sistema a través de la SSDT, tal es el caso de la función CreatePalette.



```
.text:000007FF7FD75304 ; Exported entry 1085. CreatePalette
.text:000007FF7FD75304
.text:000007FF7FD75304
.text:000007FF7FD75304
.text:000007FF7FD75304 ; HPALETTE __stdcall CreatePalette(const LOGPALETTE *lpal)
.text:000007FF7FD75304 public CreatePalette
.text:000007FF7FD75304 CreatePalette proc near
.text:000007FF7FD75304
.text:000007FF7FD75304 ; FUNCTION CHUNK AT .text:000007FF7FD8E2EA SIZE 00000012 BYTES
.text:000007FF7FD75304
.text:000007FF7FD75304 sub     rsp, 28h
.text:000007FF7FD75308 test    rcx, rcx
.text:000007FF7FD7530B jz      loc_7FF7FD8E2EA

.text:000007FF7FD75311 movzx    edx, word ptr [rcx+2]
.text:000007FF7FD75315 call     NtGdiCreatePaletteInternal

.text:000007FF7FD8E2EA ; START OF FUNCTION CHUNK FOR CreatePalette
.text:000007FF7FD8E2EA
.text:000007FF7FD8E2EA loc_7FF7FD8E2EA: ; dwErrCode
.text:000007FF7FD8E2EA mov     ecx, 57h ; 'W'
.text:000007FF7FD8E2EF call    cs:__imp_SetLastError
.text:000007FF7FD8E2F5 xor     eax, eax
.text:000007FF7FD8E2F7 jmp     loc_7FF7FD7531A
.text:000007FF7FD8E2F7 ; END OF FUNCTION CHUNK FOR CreatePalette

.text:000007FF7FD7531A
.text:000007FF7FD7531A loc_7FF7FD7531A:
.text:000007FF7FD7531A add     rsp, 28h
.text:000007FF7FD7531E netn
```

Vemos debajo que corresponde a gdi32.dll.

# Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	wingdi.h (include Windows.h)
Library	Gdi32.lib
DLL	Gdi32.dll

## See also

[Brush Functions](#)

[Brushes Overview](#)

[GetDeviceCaps](#)

Si abrimos gdi32.dll en este caso uso una versión correspondiente a Windows 7 de 64 bits.

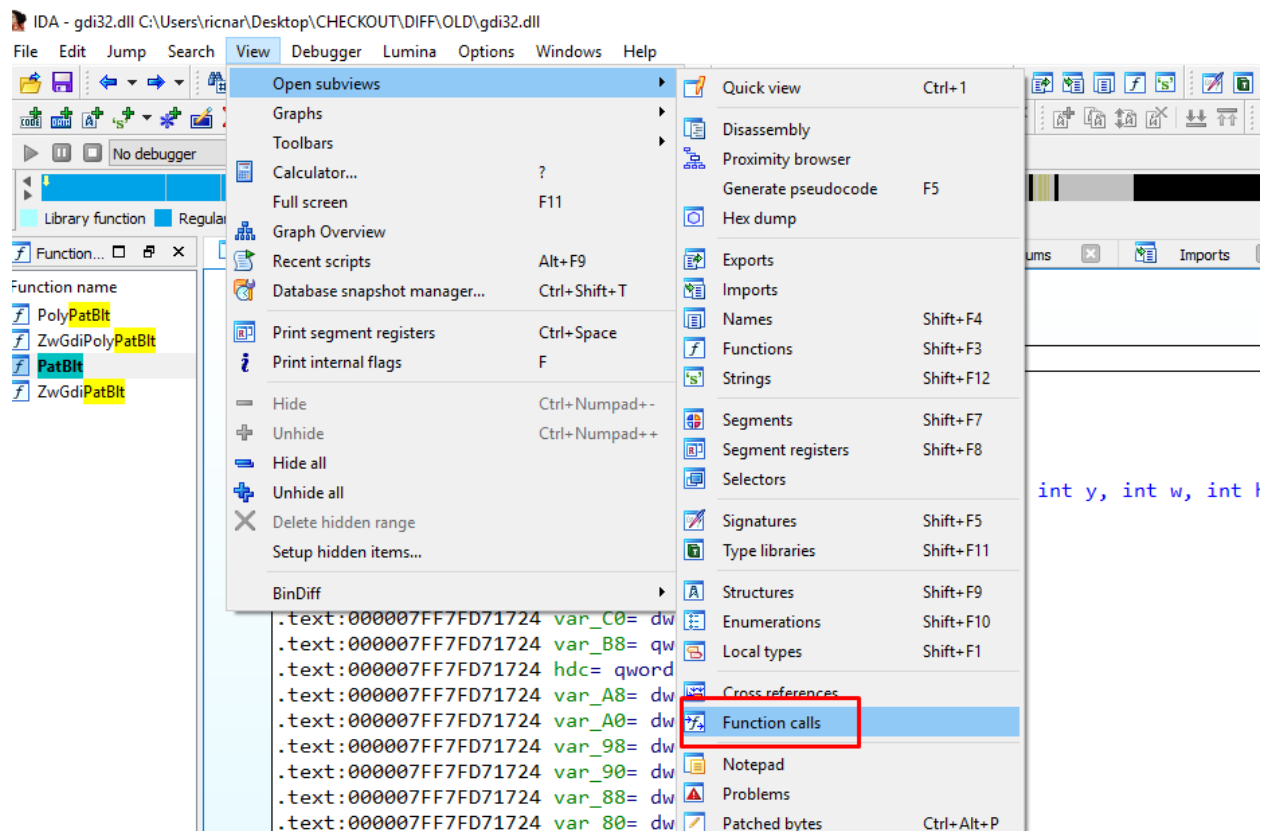
```
.text:000007FF7FD75304 ; Exported entry 1085. CreatePalette
.text:000007FF7FD75304
.text:000007FF7FD75304
.text:000007FF7FD75304
.text:000007FF7FD75304 ; HPALETTE __stdcall CreatePalette(const LOGPALETTE *lpal)
.text:000007FF7FD75304 public CreatePalette
.text:000007FF7FD75304 CreatePalette proc near
.text:000007FF7FD75304
.text:000007FF7FD75304 ; FUNCTION CHUNK AT .text:000007FF7FD8E2EA SIZE 00000012 BYTES
.text:000007FF7FD75304
.text:000007FF7FD75304 sub     rsp, 28h
.text:000007FF7FD75308 test    rcx, rcx
.text:000007FF7FD7530B jz      loc_7FF7FD8E2EA

.text:000007FF7FD75311 movsx   edx, word ptr [rcx+2]
.text:000007FF7FD75315 call    NtGdiCreatePaletteInternal

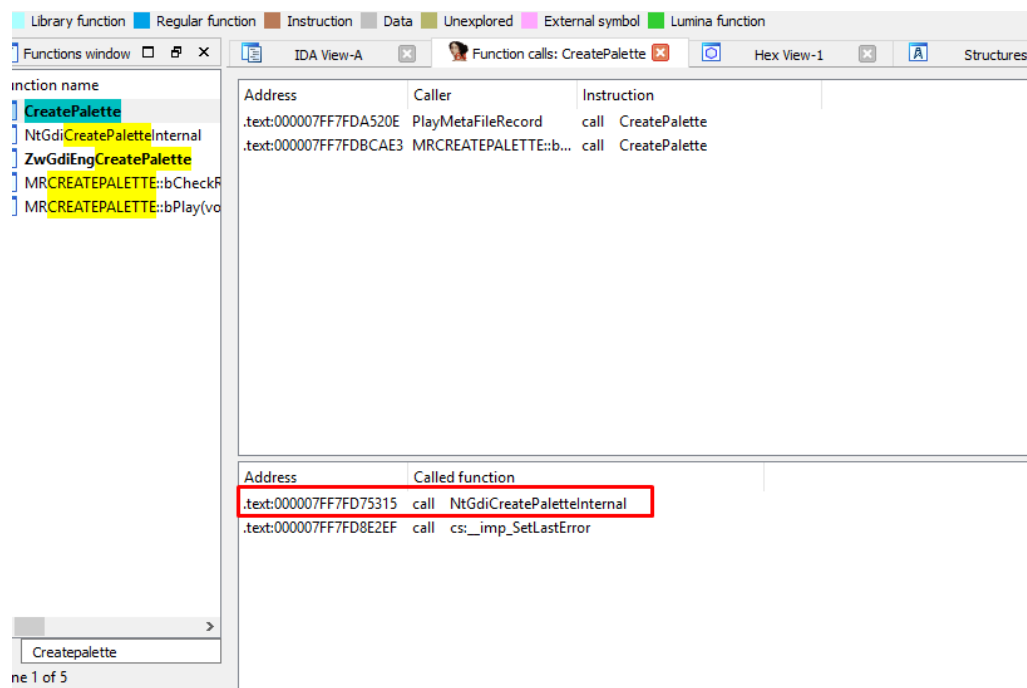
.text:000007FF7FD8E2EA ; START OF FUNCTION CHUNK FOR CreatePalette
.text:000007FF7FD8E2EA
.text:000007FF7FD8E2EA loc_7FF7FD8E2EA: ; dwErrCode
.text:000007FF7FD8E2EA mov     ecx, 57h ; 'W'
.text:000007FF7FD8E2EF call    cs:__imp_SetLastError
.text:000007FF7FD8E2F5 xor     eax, eax
.text:000007FF7FD8E2F7 jmp     loc_7FF7FD7531A
.text:000007FF7FD8E2F7 ; END OF FUNCTION CHUNK FOR CreatePalette

.text:000007FF7FD7531A
.text:000007FF7FD7531A loc_7FF7FD7531A:
.text:000007FF7FD7531A add     rsp, 28h
.text:000007FF7FD7531E retn
```

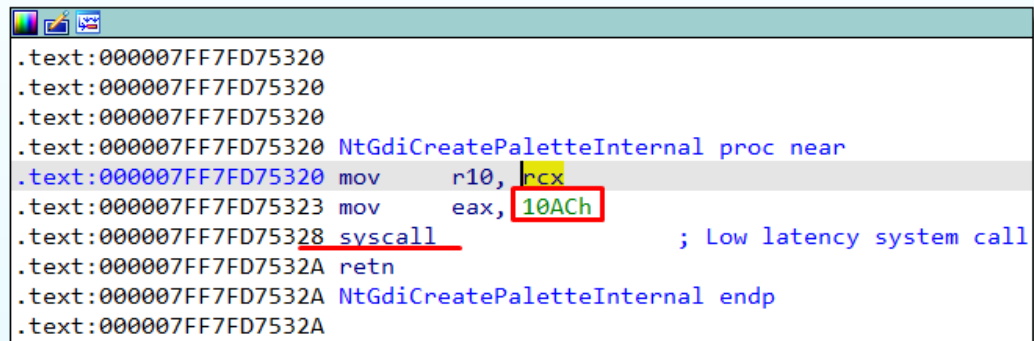
(3,5) (1135,128) 00004715 000007FF7FD75315: CreatePalette+11 (Synchronized with Hex View-1)



Vemos los call que realiza.



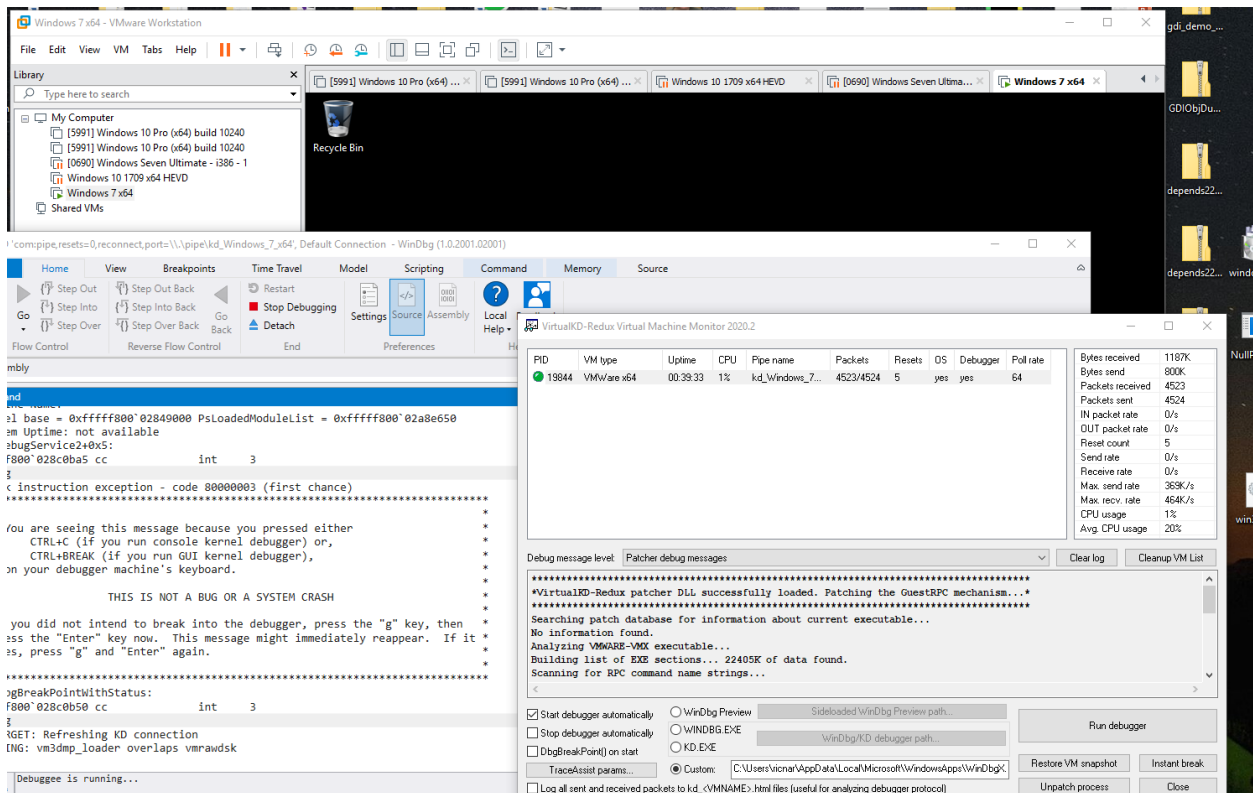
Casi siempre las funciones intermediarias llevan el prefijo ZW o NT en este caso es NT.



```
.text:000007FF7FD75320
.text:000007FF7FD75320
.text:000007FF7FD75320
.text:000007FF7FD75320 NtGdiCreatePaletteInternal proc near
.text:000007FF7FD75320 mov     r10, rcx
.text:000007FF7FD75323 mov     eax, 10ACh
.text:000007FF7FD75328 syscall                ; Low latency system call
.text:000007FF7FD7532A retn
.text:000007FF7FD7532A NtGdiCreatePaletteInternal endp
.text:000007FF7FD7532A
```

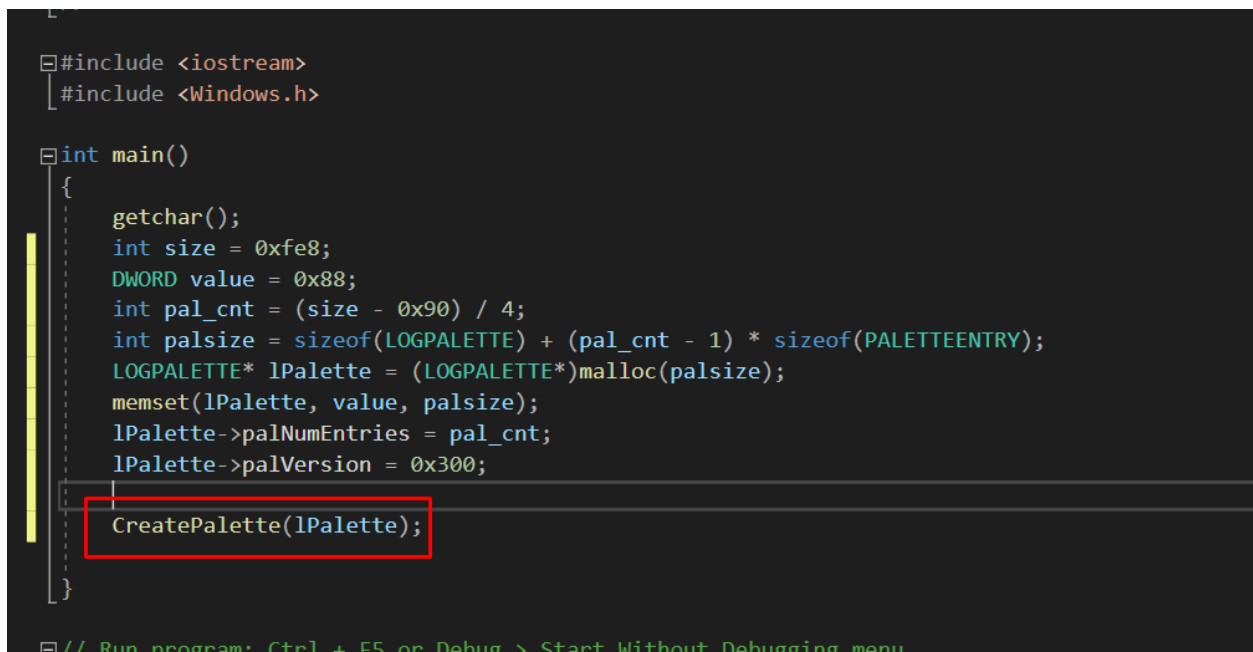
Bueno allí vemos la llamada al sistema que se realiza desde NtGdiCreatePaletteInternal, usando la instrucción syscall y con un índice en este caso 0x10ac que es el índice.

Obviamente desde un debugger en user mode no podremos entrar y tracear para continuar a ver como resuelve adonde va usando ese índice, si tenemos armado con Windbg y VMware un sistema para debugger kernel, como hemos explicado muchas veces, en este caso usando VKD ya que el target es Windows 7 de 64 bits.



Ahí esta mi sistema de Windows 7 64 bits siendo debuggeando usando VKD REDUX y Windbg preview.

Compilare un ejecutable para llamar a la función que quiero mirar.



No importa mucho esto, pero el argumento es un puntero a una estructura del tipo LOGPALETTE

12/05/2018 • 2 minutes to read

The **CreatePalette** function creates a logical palette.

## Syntax

C++

 Copy

```
HPALETTE CreatePalette(  
    const LOGPALETTE *lpal  
);
```

## Parameters

**lpal**

A pointer to a [LOGPALETTE](#) structure that contains information about the colors in the logical palette.

## Return value

If the function succeeds, the return value is a handle to a logical palette.

If the function fails, the return value is **NULL**.

## LOGPALETTE structure

12/05/2018 • 2 minutes to read

The **LOGPALETTE** structure defines a logical palette.

## Syntax

C++

 Cop

```
typedef struct tagLOGPALETTE {  
    WORD        palVersion;  
    WORD        palNumEntries;  
    PALETTEENTRY palPalEntry[1];  
} LOGPALETTE, *PLOGPALETTE, *NPLOGPALETTE, *LPLOGPALETTE;
```

## Members

**palVersion**

The version number of the system.

**palNumEntries**

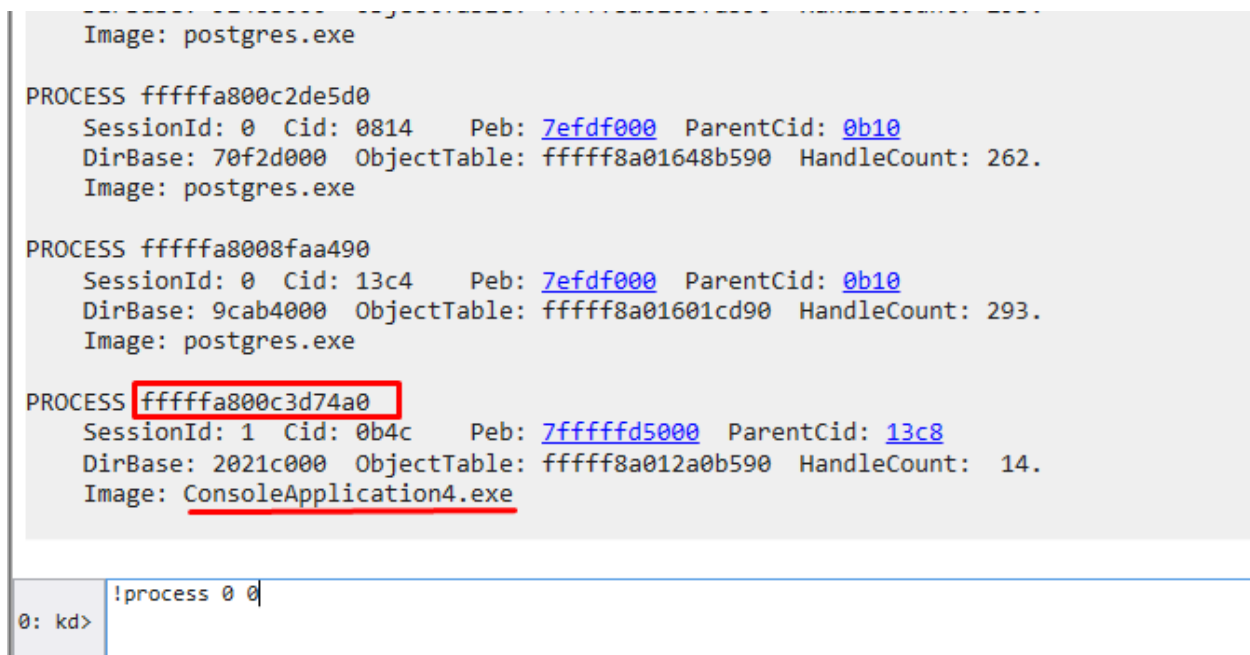
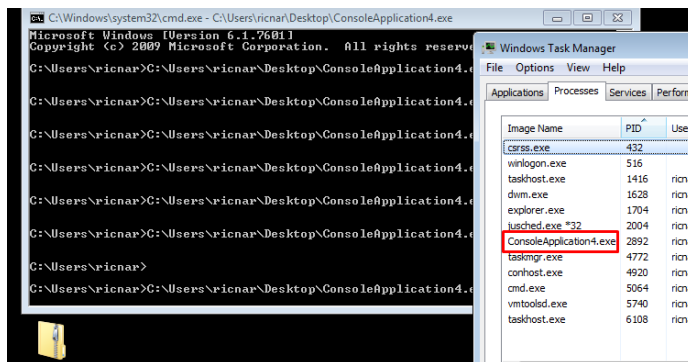
The number of entries in the logical palette.

**palPalEntry**

Specifies an array of [PALETTEENTRY](#) structures that define the color and usage of each entry in the logical palette.

Bueno eso no importa tanto para el caso, la idea es compilarlo y si se puede, llegar a entrar al sistema y mirar la SSDT.

Lo ejecuto en mi target de Windows 7 64 bits y quedara esperando en el getchar() eso me dará oportunidad para breakear el Windbg.



Obtengo el EPROCESS de mi proceso y ya puedo poner breakpoints en mi función para que se detenga. Para switchear el contexto.

```
DirBase: 2021c000 ObjectTable: fffff8a012a0b590 HandleCount: 14.  
Image: postgres.exe  
  
PROCESS fffffa800c3d74a0  
  SessionId: 1 Cid: 0b4c Peb: 7fffffd5000 ParentCid: 13c8  
  DirBase: 2021c000 ObjectTable: fffff8a012a0b590 HandleCount: 14.  
  Image: ConsoleApplication4.exe  
  
0: kd> .process /i fffffa800c3d74a0
```

Usando el **EPROCESS** de mi proceso luego **G** y cuando se detiene luego chequeo con:

**!process -1 0**

```
Image: ConsoleApplication4.exe  
  
0: kd> .process /i fffffa800c3d74a0  
You need to continue execution (press 'g' <enter>) for the context  
to be switched. When the debugger breaks in again, you will be in  
the new process context.  
0: kd> g  
Break instruction exception - code 80000003 (first chance)  
nt!DbgBreakPointWithStatus:  
fffff800`028c0b50 cc int 3  
0: kd> !process -1 0  
PROCESS fffffa800c3d74a0  
  SessionId: 1 Cid: 0b4c Peb: 7fffffd5000 ParentCid: 13c8  
  DirBase: 2021c000 ObjectTable: fffff8a012a0b590 HandleCount: 14.  
  Image: ConsoleApplication4.exe  
  
0: kd>
```

Ya estoy en mi proceso veamos la función buscada.

```
0: kd> x gdi!*CreatePalette*  
000007fe`feab5a20 gdi32!CreatePalette (CreatePalette)  
  
0: kd>
```



Cuando ejecuto el comando **u rip** me da error de acceso, eso suele ocurrir cuando uno debuggea kernel ya que el sistema no tiene toda la memoria disponible cargada, sino que hay partes de la misma que solo podemos acceder cuando las ejecuta.

```
SessionId: 0 Cid: 03bc Peb: 7fffffff0000 ParentCid: 06cc
DirBase: 1b5d0000 ObjectTable: fffff8a01667c590 HandleCount: 264.
Image: postgres.exe

1: kd> bp /p fffff800c1c1060 GDI32!CreatePalette
1: kd> u GDI32!CreatePalette
GDI32!CreatePalette:
000007fe`feab5a20 ??             ???
                                ^ Memory access error in 'u GDI32!CreatePalette'
```

Igual como dicha parte de la memoria pertenece a un módulo cargado, sabemos que, aunque no lo muestre, en el momento que ejecute la podremos ver, así que pongo un breakpoint en la función, usando el /p con el eprocess para que pare solamente cuando ese proceso llame a la función.

```
1: kd> bp /p fffff800c1c1060 GDI32!CreatePalette
1: kd> u GDI32!CreatePalette
GDI32!CreatePalette:
000007fe`feab5a20 ??             ???
                                ^ Memory access error in 'u GDI32!CreatePalette'

1: kd> g
Breakpoint 3 hit
GDI32!CreatePalette:
0033:000007fe`feab5a20 4883ec28      sub     rsp,28h

0: kd>
```

Cuando apreto ENTER en la consola en el target, para y ya veo que ahora si se puede ver el código desensamblado de la misma.

```

1: kd> g
Breakpoint 3 hit
GDI32!CreatePalette:
0033:000007fe`feab5a20 4883ec28      sub     rsp,28h
0: kd> u
GDI32!CreatePalette:
000007fe`feab5a20 4883ec28      sub     rsp,28h
000007fe`feab5a24 4885c9       test    rcx,rcx
000007fe`feab5a27 0f8403880100  je     GDI32!CreatePalette+0x9 (000007fe`feace230)
000007fe`feab5a2d 0fb75102     movzx   edx,word ptr [rcx+2]
000007fe`feab5a31 e80a000000   call    GDI32!NtGdiCreatePaletteInternal (000007fe`feab5a40)
000007fe`feab5a36 4883c428     add     rsp,28h
000007fe`feab5a3a c3          ret
000007fe`feab5a3b 90          nop

```

Por eso confirmo que hay partes de la memoria que no están disponibles siempre para listar y que el sistema las va cargando a medida que las necesita o ejecuta.

Traceo hasta el syscall con F11 (STEP IN).

```

000007fe`feab5a3b 90          nop
0: kd> t
GDI32!CreatePalette+0x4:
0033:000007fe`feab5a24 4885c9       test    rcx,rcx
0: kd> t
GDI32!CreatePalette+0x7:
0033:000007fe`feab5a27 0f8403880100  je     GDI32!CreatePalette+0x9 (000007fe`feace230)
0: kd> t
GDI32!CreatePalette+0x18:
0033:000007fe`feab5a2d 0fb75102     movzx   edx,word ptr [rcx+2]
0: kd> t
GDI32!CreatePalette+0x1c:
0033:000007fe`feab5a31 e80a000000   call    GDI32!NtGdiCreatePaletteInternal (000007fe`feab
0: kd> t
GDI32!NtGdiCreatePaletteInternal:
0033:000007fe`feab5a40 4c8bd1       mov     r10,rcx
0: kd> t
GDI32!NtGdiCreatePaletteInternal+0x3:
0033:000007fe`feab5a43 b8ac100000   mov     eax,10ACh
0: kd> t
GDI32!NtGdiCreatePaletteInternal+0x8:
0033:000007fe`feab5a48 0f05        syscall

```

```

1: kd>

```

Allí estamos, la idea sería que si apreto una vez mas f11 podría entrar desde user a kernel, parando en la primera instrucción en kernel.

```

GDI32!NtGdiCreatePaletteInternal+0xa:
0033:000007fe`feab5a4a c3          ret

```

No me deja entrar a tracear aun apretando F11 pasa por encima del SYSCALL.

Entonces como podemos hacer para parar en kernel y saber a qué función termina yendo de la SSDT.

Obviamente si sabemos a qué función ira podemos poner un breakpoint en la misma, incluso ver con K las funciones por las que paso para llegar allí desde user.

La única forma que se me ocurre es tratar de listar la SSDT y ver si mediante el índice podemos sacar a que función ira, poner un breakpoint allí y cuando pare mirar el camino que realizo.

## COMO PODEMOS LISTAR LA SSDT?

Primero buscaremos el módulo nt de kernel

```
fffff800`02849000 fffff800`02e32000  nt          (pdb symbols)      c:\symbols\
Loaded symbol image file: ntkrnlmp.exe
Image path: ntkrnlmp.exe
Image name: ntkrnlmp.exe
Browse all global symbols  functions  data
Timestamp:      Sat Apr  9 01:15:23 2011 (4D9FDD5B)
Checksum:       0054E319
ImageSize:      005E9000
File version:   6.1.7601.17592
Product version: 6.1.7601.17592
File flags:     0 (Mask 3F)
File OS:        40004 NT Win32
File type:      1.0 App
File date:      00000000.00000000
Translations:   0409.04b0
Information from resource tables:
  CompanyName:   Microsoft Corporation
  ProductName:   Microsoft® Windows® Operating System
  InternalName:  ntkrnlmp.exe
  OriginalFilename ntkrnlmp.exe
  ProductVersion: 6.1.7601.17592
  FileVersion:   6.1.7601.17592 (win7sp1_gdr.110408-1631)
  FileDescription: NT Kernel & System
  LegalCopyright: © Microsoft Corporation. All rights reserved.
```

Allí vemos su nombre ntkrnlmp.exe y si lo buscamos en la maquina target no estará.

Overall, there are four kernel image files for each revision of Windows, and two kernel image files for each Windows system. Multiprocessor or unipr selected by boot.ini or BCD option, according to the processor's features.<sup>[a]</sup>

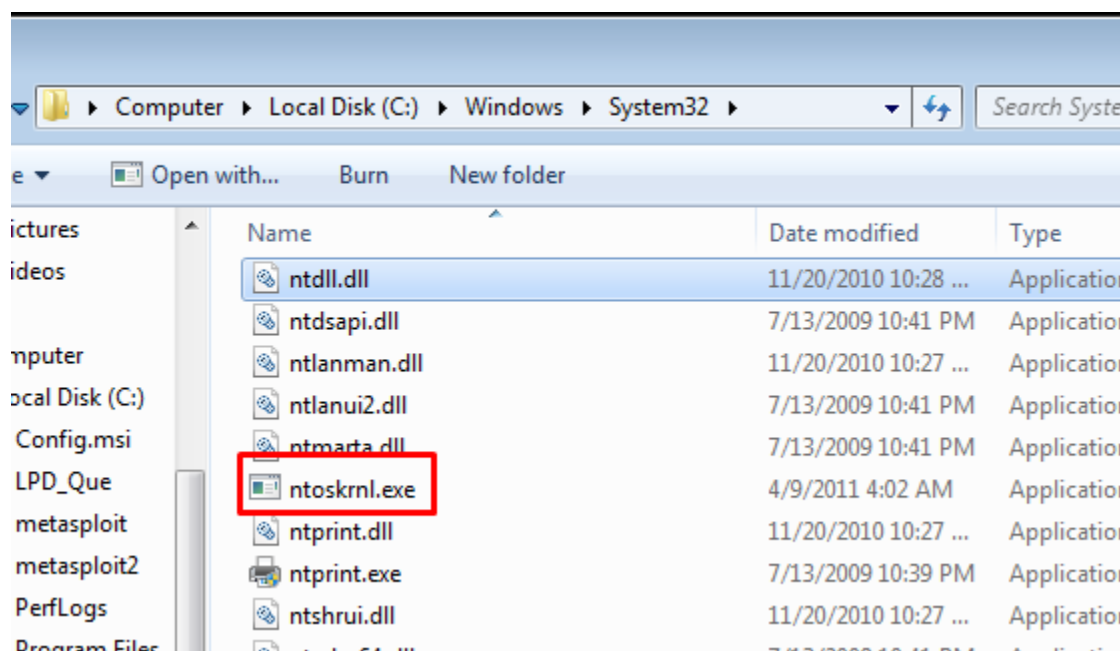
Kernel image filenames

Filename	Supports SMP	Supports PAE
ntoskrnl.exe	No	No
ntkrnlmp.exe	Yes	No
ntkrnlpa.exe	No	Yes
ntkrpamp.exe	Yes	Yes

## Notes [edit]

a. ^ On a multiprocessor system ntkrnlmp.exe is installed as ntoskrnl.exe and ntkrpamp.exe is installed as ntkrnlpa.exe .

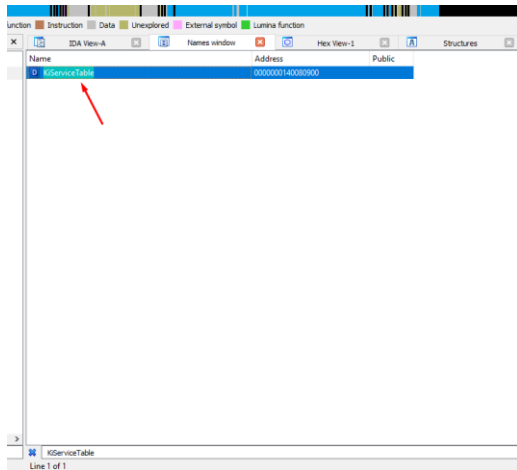
Bueno ese es el archivo a buscar, está entre los archivos ocultos en la carpeta SYSTEM32, por lo cual hay que cambiar la visibilidad para que se puedan mostrar los mismos en las opciones de carpeta.



Lo copiare a la maquina principal para abrir en IDA.

Effectively, syscalls and SSDT ( KiServiceTable ) work together as a bridge between userland API calls and their corresponding kernel routines, allowing the kernel to know which routine should be executed for a given syscall that originated in the user space.

Si busco en NAMES en IDA veo que esta KiServiceTable.



```

.text:0000001400808D8:
.text:0000001400808D8: align_1400808D8: ; DATA XREF: .pdata:0000001400808D8
.text:0000001400808D8: align 40h
.text:000000140080900: KiServiceTable dq offset NtMapUserPhysicalPagesScatter
.text:000000140080900: ; DATA XREF: KiInitializeKernel
.text:000000140080900: ; KiInitSystem+10140
.text:000000140080908: dq offset NtWaitForSingleObject
.text:000000140080910: dq offset NtCallbackReturn
.text:000000140080918: dq offset NtReadFile
.text:000000140080920: dq offset NtDeviceIoControlFile
.text:000000140080928: dq offset NtWriteFile
.text:000000140080930: dq offset NtRemoveIoCompletion
.text:000000140080938: dq offset NtReleaseSemaphore
.text:000000140080940: dq offset NtReplyWaitReceivePort
.text:000000140080948: dq offset NtReplyPort
.text:000000140080950: dq offset NtSetInformationThread
.text:000000140080958: dq offset NtSetEvent
.text:000000140080960: dq offset NtClose
.text:000000140080968: dq offset NtQueryObject
.text:000000140080970: dq offset NtQueryInformationFile
.text:000000140080978: dq offset NtOpenKey
.text:000000140080980: dq offset NtEnumerateValueKey
.text:000000140080988: dq offset NtFindAtom
.text:000000140080990: dq offset NtQueryDefaultLocale
.text:000000140080998: dq offset NtQueryKey

```

Bueno ahí está la SSDT.

Veamos si coincide hay una pagina

<https://github.com/j00ru/windows-syscalls>

Que se puede listar todos los índices de cada función de user, también veo que hay dos tablas una correspondiente a ntoskrnl y otra win32 y para cada una de ellos la versión de 64 y 32 bits.

# Windows System Call Tables

The repository contains system call tables collected from all modern and most older releases of Windows, starting with Windows NT.

Both 32-bit and 64-bit builds were analyzed, and the tables were extracted from both the core kernel image ( `ntoskrnl.exe` ) and the graphical subsystem ( `win32k.sys` ).

## Formats

The data is formatted in the CSV and JSON formats for programmatic use, and as an HTML table for manual inspection.

The HTML files are also hosted on my blog under the following links:

- ntoskrnl.exe , x86: <http://j00ru.vexillium.org/syscalls/nt/32/>
- ntoskrnl.exe , x64: <http://j00ru.vexillium.org/syscalls/nt/64/>
- win32k.sys , x86: <http://j00ru.vexillium.org/syscalls/win32k/32/>
- win32k.sys , x64: <http://j00ru.vexillium.org/syscalls/win32k/64/>

Vayamos por ahora a la de ntoskrnl.

Special thanks to: MeMek, Wandering Glitch

Layout by Metasploit Team

Enter the Syscall ID to highlight (hex):

Highlight

Show all Hide all

System Call Symbol	Windows XP	Windows Server 2003	Windows Vista	Windows Server 2008	Windows 7	Windows Server 2012	Windows 8	Windows 10
	(show)	(show)	(show)	(show)	(show)	(show)	(show)	(show)
NtAcceptConnectPort								
NtAccessCheck								
NtAccessCheckAndAuditAlarm								
NtAccessCheckByType								
NtAccessCheckByTypeAndAuditAlarm								
NtAccessCheckByTypeResultList								
NtAccessCheckByTypeResultListAndAuditAlarm								
NtAccessCheckByTypeResultListAndAuditAlarmByHandle								
NtAcquireCmpViewOwnership								
NtAcquireCrossVmMutant								
NtAcquireProcessActivityReference								
NtAddAtom								
NtAddAtomEx								
NtAddBootEntry								
NtAddDriverEntry								

Vemos que como todo array empieza con el índice cero

[illegible]

La primera entrada corresponde a la función de user NtMapUserPhysicalScatter y en IDA vemos la función correspondiente en kernel.

```

.text:00000001400808D7
.text:00000001400808D7 ; -----
.text:00000001400808D8 align_1400808D8: ; DATA XREF: .pdata:0000000140286200↓o
.text:00000001400808D8 align 40h
.text:0000000140080900 KiServiceTable dq offset NtMapUserPhysicalPagesScatter
.text:0000000140080900 ; DATA XREF: KiInitializeKernel+344↓o
.text:0000000140080900 ; KiInitSystem+101↓o
.text:0000000140080908 dq offset NtWaitForSingleObject
.text:0000000140080910 dq offset NtCallbackReturn
.text:0000000140080918 dq offset NtReadFile
.text:0000000140080920 dq offset NtDeviceIoControlFile
.text:0000000140080928 dq offset NtWriteFile
.text:0000000140080930 dq offset NtRemoveIoCompletion
.text:0000000140080938 dq offset NtReleaseSemaphore
.text:0000000140080940 dq offset NtReplyWaitReceivePort
.text:0000000140080948 dq offset NtReplyPort

```



```

.text:0000000078EA1340 ; Exported entry 341. NtMapUserPhysicalPagesScatter
.text:0000000078EA1340 ; Exported entry 1589. ZwMapUserPhysicalPagesScatter
.text:0000000078EA1340
.text:0000000078EA1340
.text:0000000078EA1340 public ZwMapUserPhysicalPagesScatter
.text:0000000078EA1340 ZwMapUserPhysicalPagesScatter proc near
.text:0000000078EA1340 mov     r10, rcx ; NtMapUserPhysicalPagesScatter
.text:0000000078EA1343 mov     eax, 0
.text:0000000078EA1348 syscall ; Low latency system call
.text:0000000078EA134A ret
.text:0000000078EA134A ZwMapUserPhysicalPagesScatter endp
.text:0000000078EA134A

```

Bueno si veo en la ntdll de user veo la llamada en este caso la llama con ZW en vez de NT pero es la misma función y con el índice cero.

Vuelvo a arrancar el ejecutable en el target y cuando queda detenido en la consola hago break en el Windbg y switcheo el contexto. (no repetiré como nuevamente)

```
DirBase: 23387f000 ObjectTable: fffff8a013470870 HandleCount: 284.  
Image: postgres.exe
```

```
PROCESS fffff8007ab0060
```

```
SessionId: 1 Cid: 0a70 Peb: 7fffffffdb000 ParentCid: 13c8  
DirBase: 8d4dd000 ObjectTable: fffff8a001e18590 HandleCount: 14.  
Image: ConsoleApplication4.exe
```

```
0: kd> .process /i fffff8007ab0060
```

You need to continue execution (press 'g' <enter>) for the context to be switched. When the debugger breaks in again, you will be in the new process context.

```
0: kd> g
```

Break instruction exception - code 80000003 (first chance)

```
nt!DbgBreakPointWithStatus:
```

```
fffff800`028c0b50 cc int 3
```

```
0: kd> !process -1 0
```

```
PROCESS fffff8007ab0060
```

```
SessionId: 1 Cid: 0a70 Peb: 7fffffffdb000 ParentCid: 13c8  
DirBase: 8d4dd000 ObjectTable: fffff8a001e18590 HandleCount: 14.  
Image: ConsoleApplication4.exe
```

Puedo ver en windbg que hay dos SSDT una para ntoskrnl que acabamos de ver y otra para win32ksys.

Looking at the Windbg outputs above, we see that there is one clearly identified SST table in the case of **nt!KeServiceDescriptorTable** and two in the case of **nt!KeServiceDescriptorTableShadow**. Although we see more data in there, from available information, we can only state that from the possible 4 SST entries, the **nt!KeServiceDescriptorTable** uses only the first one - it describes the SSDT for the Windows Native APIs exported by **ntoskrnl.exe**. The **nt!KeServiceDescriptorTableShadow** uses 2 SST entries, the first is a copy of the **nt!KiServiceTable** from **nt!KeServiceDescriptorTable**, the second one, **win32k!W32pServiceTable**, describes the SSDT for the User and GDI routines exported by **win32k.sys**.

```
0: kd> dps nt!keserviceDescriptorTable
fffff800`02af8940 fffff800`028c9900 nt!KiServiceTable
fffff800`02af8948 00000000`00000000
fffff800`02af8950 00000000`00000191
fffff800`02af8958 fffff800`028ca58c nt!KiArgumentTable
fffff800`02af8960 00000000`00000000
fffff800`02af8968 00000000`00000000
fffff800`02af8970 00000000`00000000
fffff800`02af8978 00000000`00000000
fffff800`02af8980 fffff800`028c9900 nt!KiServiceTable
fffff800`02af8988 00000000`00000000
fffff800`02af8990 00000000`00000191
fffff800`02af8998 fffff800`028ca58c nt!KiArgumentTable
fffff800`02af89a0 fffff960`00171f00 win32k!W32pServiceTable
fffff800`02af89a8 00000000`00000000
fffff800`02af89b0 00000000`0000033b
fffff800`02af89b8 fffff960`00173c1c win32k!W32pArgumentTable
```



Allí están las 2 si abrimos una copia de win32ksys copiado de la maquina target en IDA veremos esta segunda SSDT.

```

.text:FFFFFF97FFF0D5F4A align_FFFFFF97FFF0D5F4A: ; DATA XREF: .pdata:FFFFFF97FFF3002FC↓o
.text:FFFFFF97FFF0D5F4A align 100h
.text:FFFFFF97FFF0D6000 ; Exported entry 224. W32pServiceTable
.text:FFFFFF97FFF0D6000 public W32pServiceTable
.text:FFFFFF97FFF0D6000 W32pServiceTable dq offset NtUserGetThreadState
.text:FFFFFF97FFF0D6000 ; DATA XREF: DriverEntry+290↓o
.text:FFFFFF97FFF0D6008 dq offset NtUserPeekMessage
.text:FFFFFF97FFF0D6010 dq offset NtUserCallOneParam
.text:FFFFFF97FFF0D6018 dq offset NtUserGetKeyState
.text:FFFFFF97FFF0D6020 dq offset NtUserInvalidateRect
.text:FFFFFF97FFF0D6028 dq offset NtUserCallNoParam
.text:FFFFFF97FFF0D6030 dq offset NtUserGetMessage
.text:FFFFFF97FFF0D6038 dq offset NtUserMessageCall
.text:FFFFFF97FFF0D6040 dq offset NtGdiBitBlt
.text:FFFFFF97FFF0D6048 dq offset NtGdiGetCharSet
.text:FFFFFF97FFF0D6050 dq offset NtUserGetDC
.text:FFFFFF97FFF0D6058 dq offset NtGdiSelectBitmap
.text:FFFFFF97FFF0D6060 dq offset NtUserWaitMessage
.text:FFFFFF97FFF0D6068 dq offset NtUserTranslateMessage
.text:FFFFFF97FFF0D6070 dq offset NtUserGetProp
.text:FFFFFF97FFF0D6078 dq offset NtUserPostMessage
.text:FFFFFF97FFF0D6080 dq offset NtUserQueryWindow
.text:FFFFFF97FFF0D6088 dq offset NtUserTranslateAccelerator
.text:FFFFFF97FFF0D6090 dq offset NtGdiFlush
.text:FFFFFF97FFF0D6098 dq offset NtUserRedrawWindow
.text:FFFFFF97FFF0D60A0 dq offset NtUserWindowFromPoint
.text:FFFFFF97FFF0D60A8 dq offset NtUserCallMsgFilter
.text:FFFFFF97FFF0D60B0 dq offset NtUserValidateTimerCallback
.text:FFFFFF97FFF0D60B8 dq offset NtUserBeginPaint
.text:FFFFFF97FFF0D60C0 dq offset NtUserSetTimer
.text:FFFFFF97FFF0D60C8 dq offset NtUserEndPaint

000D5400 FFFFFFF97FFF0D6000: .text:W32pServiceTable (Synchronized with Hex View-1)

```

También vimos que la pagina donde lista tiene otra página separada para esta otra SSDT.

The data is formatted in the CSV and JSON formats for programmatic use, and as an HTML

The HTML files are also hosted on my blog under the following links:

- ntoskrnl.exe , x86: <http://j00ru.vexillium.org/syscalls/nt/32/>
- ntoskrnl.exe , x64: <http://j00ru.vexillium.org/syscalls/nt/64/>
- win32k.sys , x86: <http://j00ru.vexillium.org/syscalls/win32k/32/>
- win32k.sys , x64: <http://j00ru.vexillium.org/syscalls/win32k/64/> 

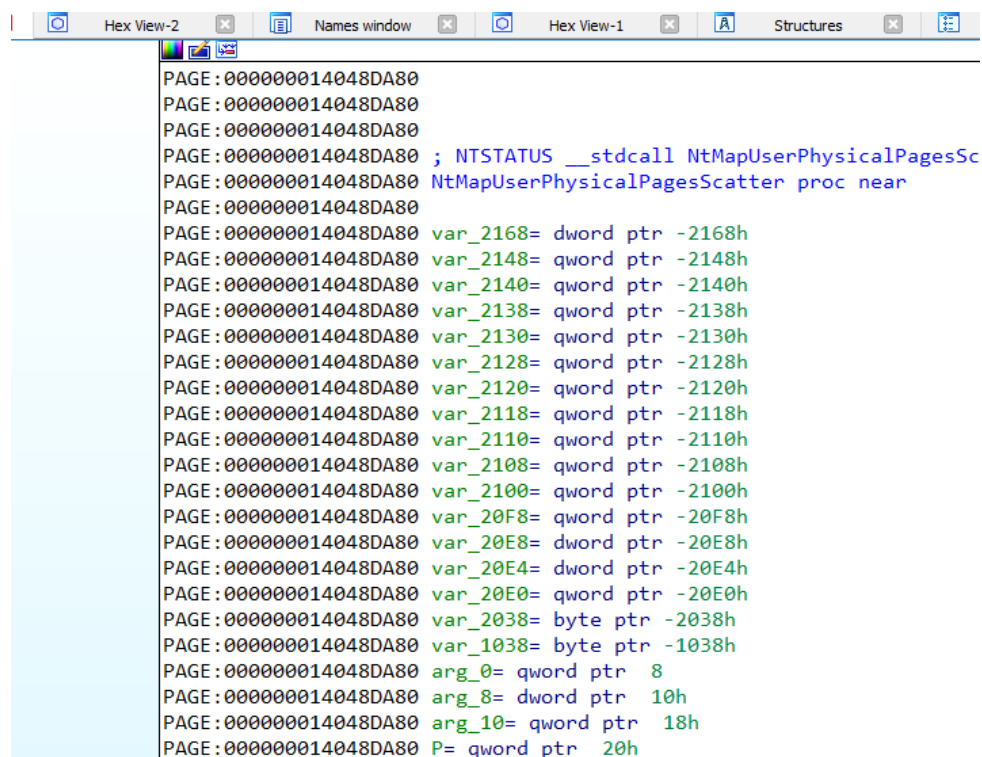
Vemos que el índice mas bajo de esta tabla segunda tabla es 0x1000, ahora como podemos convertir la información que nos lista Windbg para obtener la dirección donde saltaría en cualquiera de ambas tablas, ya sabemos que índices menores que 0x1000 corresponden a la tabla de ntoskrnl y mayores a la de win32k.

En 64 bits la cosa es un poco mas compleja (en 32 bits es mucho más sencillo luego compararemos)

Veamos como podemos hallar en la primera tabla de ntoskrnl la dirección de la primera función, el método se aplica a cualquier función de cualquiera de las dos tablas.

```
.text:00000001400808D7 ; -----
.text:00000001400808D8 align_1400808D8: ; DATA XREF: .pdata:0000000140286200↓o
.text:00000001400808D8 align 40h
.text:0000000140080900 KiServiceTable dq offset NtMapUserPhysicalPagesScatter
.text:0000000140080900 ; DATA XREF: KiInitializeKernel+344↓o
.text:0000000140080900 ; KiInitSystem+101↓o
.text:0000000140080908 dq offset NtWaitForSingleObject
.text:0000000140080910 dq offset NtCallbackReturn
.text:0000000140080918 dq offset NtReadFile
.text:0000000140080920 dq offset NtDeviceIoControlFile
.text:0000000140080928 dq offset NtWriteFile
```

Bueno esa es la primera entrada de esta SSDT correspondiente al índice 0, si hago doble click en IDA me muestra la función donde saltaría.



```
Hex View-2 | Names window | Hex View-1 | Structures
PAGE:000000014048DA80
PAGE:000000014048DA80
PAGE:000000014048DA80
PAGE:000000014048DA80 ; NTSTATUS __stdcall NtMapUserPhysicalPagesSc
PAGE:000000014048DA80 NtMapUserPhysicalPagesScatter proc near
PAGE:000000014048DA80
PAGE:000000014048DA80 var_2168= dword ptr -2168h
PAGE:000000014048DA80 var_2148= qword ptr -2148h
PAGE:000000014048DA80 var_2140= qword ptr -2140h
PAGE:000000014048DA80 var_2138= qword ptr -2138h
PAGE:000000014048DA80 var_2130= qword ptr -2130h
PAGE:000000014048DA80 var_2128= qword ptr -2128h
PAGE:000000014048DA80 var_2120= qword ptr -2120h
PAGE:000000014048DA80 var_2118= qword ptr -2118h
PAGE:000000014048DA80 var_2110= qword ptr -2110h
PAGE:000000014048DA80 var_2108= qword ptr -2108h
PAGE:000000014048DA80 var_2100= qword ptr -2100h
PAGE:000000014048DA80 var_20F8= qword ptr -20F8h
PAGE:000000014048DA80 var_20E8= dword ptr -20E8h
PAGE:000000014048DA80 var_20E4= dword ptr -20E4h
PAGE:000000014048DA80 var_20E0= qword ptr -20E0h
PAGE:000000014048DA80 var_2038= byte ptr -2038h
PAGE:000000014048DA80 var_1038= byte ptr -1038h
PAGE:000000014048DA80 arg_0= qword ptr 8
PAGE:000000014048DA80 arg_8= dword ptr 10h
PAGE:000000014048DA80 arg_10= qword ptr 18h
PAGE:000000014048DA80 P= qword ptr 20h
```

Obviamente sabiendo el nombre, podríamos ver en Windbg la dirección, lo haremos para chequear si nuestras cuentas dan bien, pero lo hallaremos calculando.

```
fffff800`02af89b8 fffff960`00173c1c win32k!W32pArgumentTable
0: kd> x nt!*!NtMapUserPhysicalPagesScatter
00000000`77a51340 ntdll!NtMapUserPhysicalPagesScatter (NtMapUserPhysicalPagesScatter)
fffff800`02cd6a80 nt!NtMapUserPhysicalPagesScatter (NtMapUserPhysicalPagesScatter)
```

Esa es la dirección veamos lo que muestra Windbg en la tabla en el primer campo.

```
0: kd> dps KiServiceTable
fffff800`028c9900 02f51500`040d1800
fffff800`028c9908 02e82f05`fff6f600
fffff800`028c9910 03112305`031a0306
fffff800`028c9918 02b49d00`02bb5301
fffff800`028c9920 03dcf200`03128540
fffff800`028c9928 02e78800`02c7ff00
```

Como calculamos desde allí la dirección de la función, veamos.

Debemos usar la parte baja o sea en este caso 0x40d1800

Veamos como se hace:

Bits 4-31 correspond to the relative address to the base of the `nt!KiServiceTable`. Bits 0-3 are related to the number of arguments and will not be used here.

To obtain the function address, we shift the value 4 bits to the right and add the result to the table base linear address:

```
>>> hex(0xfffff800028c9900 + (0x40d1800 >> 4))
'0xfffff80002cd6a80L'
```

Así que con la parte baja de la entrada haciendo shift con 4 y sumándole la base de la tabla llegamos a obtener la dirección de la función.

```
hex(0xfffff800028c9900 + (0x40d1800 >> 4))
```

```
'0xfffff80002cd6a80L'
```

```
0: kd> u 0xfffff80002cd6a80
nt!NtMapUserPhysicalPagesScatter:
fffff800`02cd6a80 48895c2408      mov     qword ptr [rsp+8],rbx
fffff800`02cd6a85 4c89442418      mov     qword ptr [rsp+18h],r8
fffff800`02cd6a8a 55             push    rbp
fffff800`02cd6a8b 56             push    rsi
fffff800`02cd6a8c 57             push    rdi
fffff800`02cd6a8d 4154           push    r12
fffff800`02cd6a8f 4155           push    r13
fffff800`02cd6a91 4156           push    r14
```

Coincide

```
PAGE:0000000014048DA80
PAGE:0000000014048DA80 ; NTSTATUS __stdcall NtMapUserPhysicalPagesScatter
PAGE:0000000014048DA80 NtMapUserPhysicalPagesScatter proc near
PAGE:0000000014048DA80
PAGE:0000000014048DA80 var_2168= dword ptr -2168h
PAGE:0000000014048DA80 var_2148= qword ptr -2148h
PAGE:0000000014048DA80 var_2140= qword ptr -2140h
PAGE:0000000014048DA80 var_2138= qword ptr -2138h
PAGE:0000000014048DA80 var_2130= qword ptr -2130h
PAGE:0000000014048DA80 var_2128= qword ptr -2128h
PAGE:0000000014048DA80 var_2120= qword ptr -2120h
PAGE:0000000014048DA80 var_2118= qword ptr -2118h
PAGE:0000000014048DA80 var_2110= qword ptr -2110h
PAGE:0000000014048DA80 var_2108= qword ptr -2108h
PAGE:0000000014048DA80 var_2100= qword ptr -2100h
PAGE:0000000014048DA80 var_20F8= qword ptr -20F8h
PAGE:0000000014048DA80 var_20E8= dword ptr -20E8h
PAGE:0000000014048DA80 var_20E4= dword ptr -20E4h
PAGE:0000000014048DA80 var_20E0= qword ptr -20E0h
PAGE:0000000014048DA80 var_2038= byte ptr -2038h
PAGE:0000000014048DA80 var_1038= byte ptr -1038h
PAGE:0000000014048DA80 arg_0= qword ptr 8
PAGE:0000000014048DA80 arg_8= dword ptr 10h
PAGE:0000000014048DA80 arg_10= qword ptr 18h
PAGE:0000000014048DA80 P= qword ptr 20h
PAGE:0000000014048DA80
PAGE:0000000014048DA80 mov     [rsp+arg_0], rbx
PAGE:0000000014048DA85 mov     [rsp+arg_10], r8
PAGE:0000000014048DA8A push    rbp
PAGE:0000000014048DA8B push    rsi
PAGE:0000000014048DA8C push    rdi
PAGE:0000000014048DA8D push    r12
```

Ahora esto parece fácil porque es la primera entrada veamos si lo podemos propagar para cualquier entrada.

Tomemos un numero arbitrario de índice digamos el 0x40.

Con este comando obtendremos a partir del índice directamente la parte baja que necesitamos

```
dd /c 1 nt!KiServiceTable L(INDICE +1)
```

Ese comando listará las entradas y la última que muestre será la buscada, hay que sumarles uno al índice porque eso le dice a Windbg cuantas entradas listar y para listar la primera no le podemos pasar cero, siempre hay que sumarle uno, en el caso que vimos anteriormente, para el índice 0.

```

0: kd> dd /c 1 nt!KiServiceTable L(0+1)
fffff800`028c9900 040d1800

```

Para el caso 0x40.

```

Command
fffff800`028c99a8 0411e600
fffff800`028c99ac 02663e05
fffff800`028c99b0 02d26101
fffff800`028c99b4 02da4b00
fffff800`028c99b8 02af8e00
fffff800`028c99bc 02b6f102
fffff800`028c99c0 02e794c2
fffff800`028c99c4 02f52640
fffff800`028c99c8 02e80207
fffff800`028c99cc 030918c0
fffff800`028c99d0 03186000
fffff800`028c99d4 0411a001
fffff800`028c99d8 02d90e06
fffff800`028c99dc 02a56101
fffff800`028c99e0 02d54a40
fffff800`028c99e4 02d84803
fffff800`028c99e8 02da6900
fffff800`028c99ec 02e8df80
fffff800`028c99f0 02a57801
fffff800`028c99f4 02e06a40
fffff800`028c99f8 02b3ae02
fffff800`028c99fc 02a48d82
fffff800`028c9a00 0000e400

```

```

0: kd> dd /c 1 nt!KiServiceTable L(0x40+1)

```

Veamos a que función corresponde.

```

fffff800`028c99f8 02b3ae02
fffff800`028c99fc 02a48d82
fffff800`028c9a00 0000e400

```

```

0: kd> dd /c 1 nt!KiServiceTable L(0x40+1)

```



```

fffff800`028c99fc  02a48d82
fffff800`028c9a00  0000e400
0: kd> u 0xfffff800028ca740
nt!NtContinue:
fffff800`028ca740 488b9dc000000000 mov     rbx,qword ptr [rbp+0C0h]
fffff800`028ca747 488bbdc800000000 mov     rdi,qword ptr [rbp+0C8h]
fffff800`028ca74e 488bb5d000000000 mov     rsi,qword ptr [rbp+0D0h]
fffff800`028ca755 4833c0             xor     rax,rax
fffff800`028ca758 488945b0           mov     qword ptr [rbp-50h],rax
fffff800`028ca75c 4881ec38010000    sub     rsp,138h
fffff800`028ca763 488d842400010000 lea     rax,[rsp+100h]
fffff800`028ca76b 0f29742430         movaps  xmmword ptr [rsp+30h],xmm6

```

Coincide perfectamente.

En 32 bits no es tan problemático porque no hay que realizar ninguna cuenta.

### A 32-bit Example

To clarify a bit the ideas, let's use **Windbg** once again and imagine a User Mode call to **ntdll!NtCreateFile** (**ntdll.dll** is a User Mode DLL containing, among other things, dispatch stubs to Kernel Mode system services):

```

kd> u ntdll!NtCreateFile
ntdll!NtCreateFile:
77ab3250 b875010000      mov     eax,175h
77ab3255 e803000000      call   ntdll!NtCreateFile+0xd (77ab325d)
77ab325a c22c00          ret     2Ch
77ab325d 8bd4            mov     edx,esp
77ab325f 0f34            sysenter

```

Hide Copy Code

As you see, the Dispatch ID is 0x175. It will be resolved in the **nt!KiServiceTable** SSDT to the item in the 0x176th position (the list is zero based), which is **nt!NtCreateFile**.

```

kd> dps nt!KiServiceTable L176
8190f20c 818c573a nt!NtAccessCheck
8190f210 818cbfd8 nt!NtWorkerFactoryWorkerReady
8190f214 81b033b8 nt!NtAcceptConnectPort
.....
190f7d8 81ad7b18 nt!NtCreateTimer2
8190f7dc 81af323a nt!NtCreateIoCompletion
8190f7e0 81a66958 nt!NtCreateFile

```

Hide Copy Code

Vemos que se accede en la tabla directamente sin hacer cuentas a la dirección de la función buscada.

Ahora tratemos de aplicar lo mismo a la otra tabla, la de **win32k.sys** que esta en la otra pagina y con nuestro ejemplo, en el que aún estamos detenidos.



```

.text:000007FF7FD75320
.text:000007FF7FD75320
.text:000007FF7FD75320
.text:000007FF7FD75320 NtGdiCreatePaletteInternal proc near
.text:000007FF7FD75320 mov     r10, rcx
.text:000007FF7FD75323 mov     eax, 10ACh
.text:000007FF7FD75328 syscall                ; Low latency system call
.text:000007FF7FD7532A retn
.text:000007FF7FD7532A NtGdiCreatePaletteInternal endp
.text:000007FF7FD7532A

```

Recordemos que en esta otra tabla el mínimo índice era 0x1000 tenemos que tener en cuenta eso, listemos la tabla en windbg.

J:\Users\ricnar\Desktop\CHECKOUT\DIFF\win32k.sys.i64

Debugger Lumina Options Windows Help

Instruction Data Unexplored External symbol Lumina function

IDA View-A Hex View-1 Structures Enums Imports

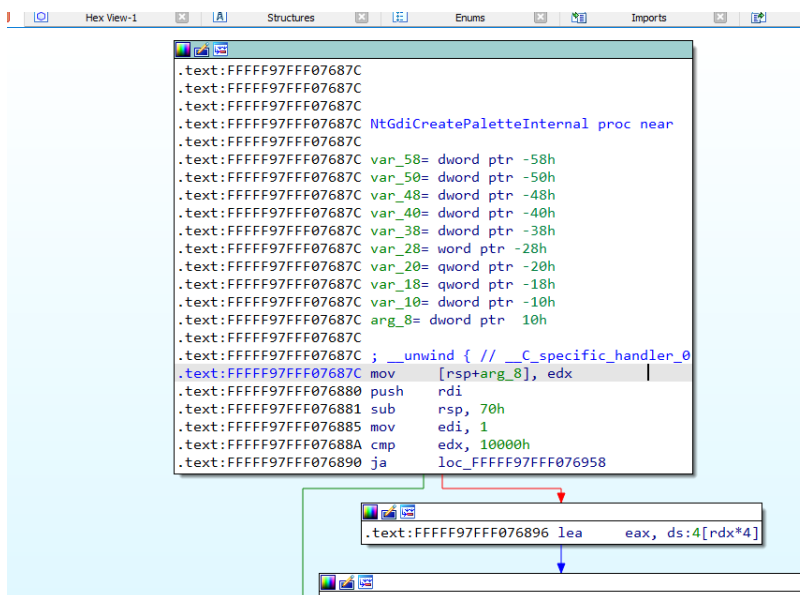
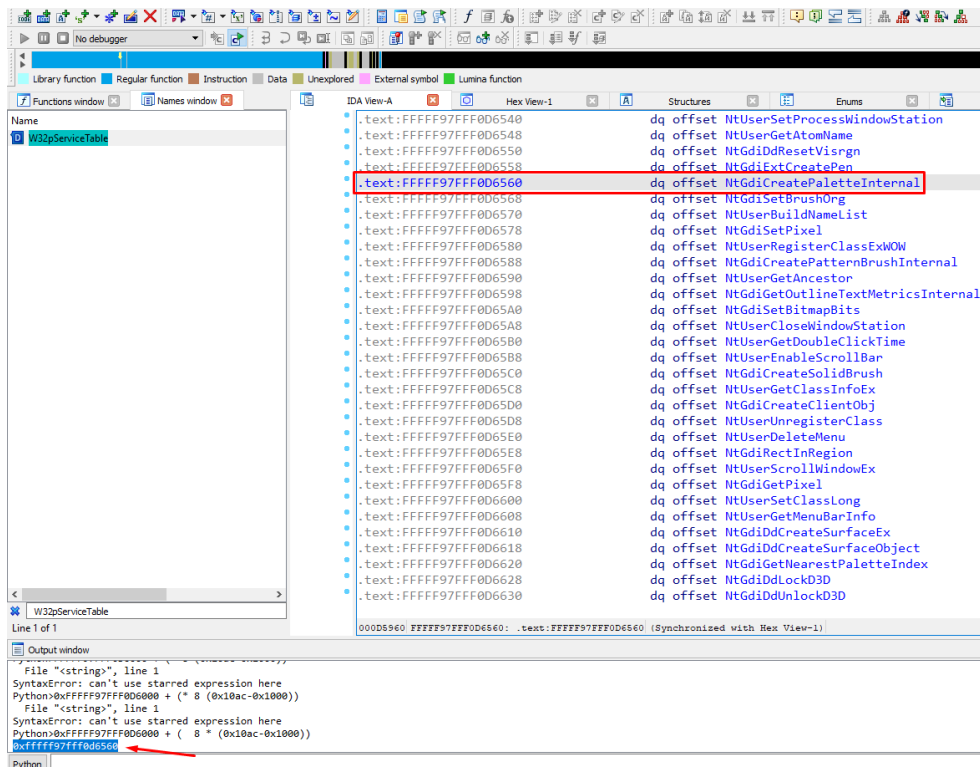
```

.text:FFFFFF97FFF0D5F4A align_FFFF97FFF0D5F4A: ; DATA XREF: .pdata:FFFFFF97FFF30021
.text:FFFFFF97FFF0D5F4A align 100h
.text:FFFFFF97FFF0D6000 ; Exported entry 224. W32pServiceTable
.text:FFFFFF97FFF0D6000 public W32pServiceTable
.text:FFFFFF97FFF0D6000 W32pServiceTable dq offset NtUserGetThreadState ; DATA XREF: DriverEntry+29040
.text:FFFFFF97FFF0D6000
.text:FFFFFF97FFF0D6008 dq offset NtUserPeekMessage
.text:FFFFFF97FFF0D6010 dq offset NtUserCallOneParam
.text:FFFFFF97FFF0D6018 dq offset NtUserGetKeyState
.text:FFFFFF97FFF0D6020 dq offset NtUserInvalidateRect
.text:FFFFFF97FFF0D6028 dq offset NtUserCallNoParam
.text:FFFFFF97FFF0D6030 dq offset NtUserGetMessage
.text:FFFFFF97FFF0D6038 dq offset NtUserMessageCall
.text:FFFFFF97FFF0D6040 dq offset NtGdiBitBlt
.text:FFFFFF97FFF0D6048 dq offset NtGdiGetCharSet
.text:FFFFFF97FFF0D6050 dq offset NtUserGetDC
.text:FFFFFF97FFF0D6058 dq offset NtGdiSelectBitmap
.text:FFFFFF97FFF0D6060 dq offset NtUserWaitMessage
.text:FFFFFF97FFF0D6068 dq offset NtUserTranslateMessage
.text:FFFFFF97FFF0D6070 dq offset NtUserGetProp

```

En IDA multiplicamos por 8 sin olvidar de restarle los 0x1000 al índice.





En windbg buscamos la tabla.

```

0: kd> x win32k!*W32pServiceTable*
fffff960`00171f00 win32k!W32pServiceTable (W32pServiceTable)

```

Como esta pertenece a win32k buscamos allí.

```
0: kd> dd win32k!W32pServiceTable
fffff960`00171f00 fff3a740 fff0b501 000206c0 001021c0
fffff960`00171f10 00096000 00022640 fff9a900 ffde0b03
fffff960`00171f20 ffb7a7c7 00fc5200 ffed2e80 ffe50e00
fffff960`00171f30 000c58c0 000af600 000e8bc0 fffeb300
fffff960`00171f40 ffb1b480 0004ec80 ffa53180 000b9480
fffff960`00171f50 000b2500 000fc200 00037cc0 000b3e40
fffff960`00171f60 ffb1f2c0 000b58c0 000a1440 ffb28600
fffff960`00171f70 ffa27303 ffa80500 0012b300 00094080
```

Veamos si podemos aplicar la formula y ahora hay que sumarle la base de esta tabla.

```
>>> hex(0xfffff96000171f00 + (0xffa261c0 >> 4))
'0xfffff9601011451cL'
```

Me da esa dirección que puedo comprobar en Windbg.

```
0: kd> u 0xfffff9601011451c
fffff960`1011451c ??             ???
^ Memory access error in 'u 0xfffff9601011451c'
0: kd> x win32k!NtGdiCreatePaletteInternal
fffff960`0011451c win32k!NtGdiCreatePaletteInternal (NtGdiCreatePaletteInternal)
```

Vemos que, aunque no pueda listarla como la vez anterior, dicha dirección corresponde a la función encontrada.

Le pondré un breakpoint.

```
1: kd> bp win32k!NtGdiCreatePaletteInternal
WARNING: Software breakpoints on session addresses can cause bugchecks.
Use hardware execution breakpoints (ba e) if possible.
1: kd> hc*
1: kd> ba e1 win32k!NtGdiCreatePaletteInternal
```

Le doy G y corro el ejecutable en el target y parara.

```
Breakpoint 0 hit
win32k!NtGdiCreatePaletteInternal:
fffff960`0011451c 89542410      mov     dword ptr [rsp+10h],edx
1: kd> !process -1 0
PROCESS fffffa8008dccb30
  SessionId: 1 Cid: 0dc0 Peb: 7fffffd8000 ParentCid: 13c8
  DirBase: 1e7e67000 ObjectTable: fffff8a0100db720 HandleCount: 14.
  Image: ConsoleApplication4.exe
```

Podría haber usado el eprocess para filtrar, pero no es tan común que se llame a esta función de kernel muy seguido, así que igual paro siendo llamada por mi proceso ahí vemos que estamos en el mismo.

Este era nuestro objetivo y lo logramos obtener la función donde saltaría desde user, usando el índice para calcularla.

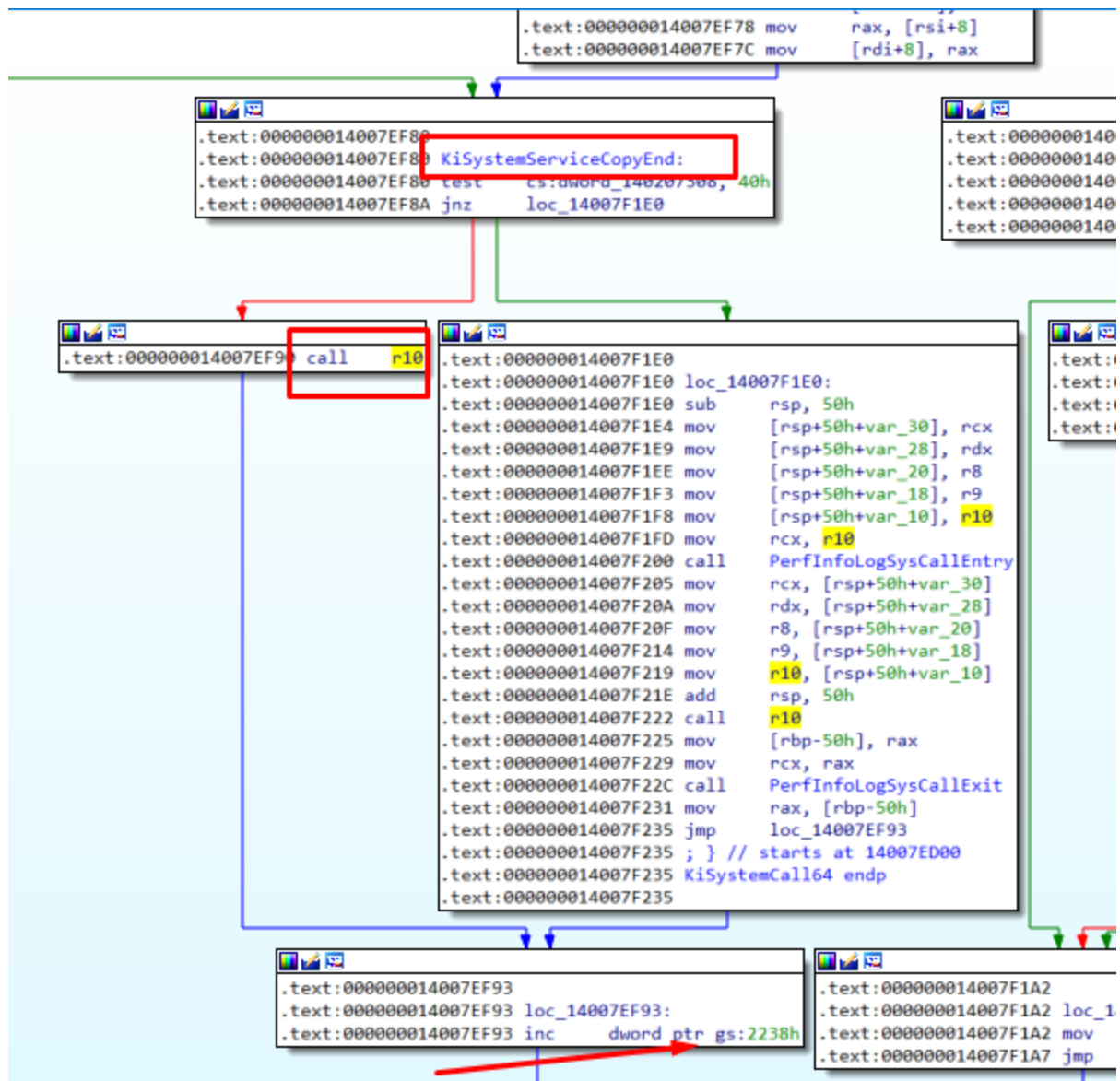
Chusmeemos un poco mas.

Allí vemos el call stack como se origino en la llamada de user CreatePallete, luego entra en user a NtGdiCreatePaletteInternal, veamos donde sigue cuando entra en el syscall en el IDA.

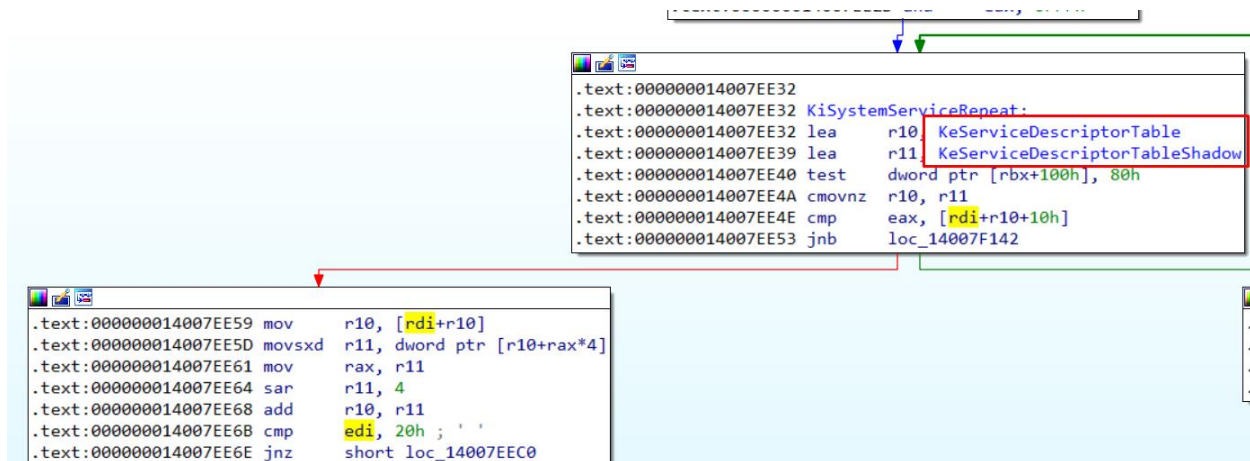
```
1: kd> k
# Child-SP          RetAddr           Call Site
00 fffff880`0701fc18 fffff800`028c7f93 win32k!NtGdiCreatePaletteInternal
01 fffff880`0701fc20 000007fe`feab5a4a nt!KiSystemServiceCopyEnd+0x13
02 00000000`0024f998 000007fe`feab5a36 GDI32!NtGdiCreatePaletteInternal+0xa
03 00000000`0024f9a0 00000001`3f1c1091 GDI32!CreatePalette+0x21
04 00000000`0024f9d0 00000001`3f1d7060 0x00000001`3f1c1091
05 00000000`0024f9d8 00000000`00378420 0x00000001`3f1d7060
06 00000000`0024f9e0 00000001`3f1ce2b0 0x378420
07 00000000`0024f9e8 00000001`3f1c14c5 0x00000001`3f1ce2b0
08 00000000`0024f9f0 000003d6`00000f5c 0x00000001`3f1c14c5
09 00000000`0024f9f8 00000088`00000fe8 0x000003d6`00000f5c
0a 00000000`0024fa00 00000000`00381120 0x00000088`00000fe8
0b 00000000`0024fa08 00000001`3f1ce2b8 0x381120
0c 00000000`0024fa10 00000000`00000000 0x00000001`3f1ce2b8
```

```
1: kd> u nt!KiSystemServiceCopyEnd+0x13
nt!KiSystemServiceCopyEnd+0x13:
fffff800`028c7f93 65ff042538220000 inc     dword ptr gs:[2238h]
nt!KiSystemServiceExit:
fffff800`028c7f9b 488b9dc0000000 mov     rbx,qword ptr [rbp+0C0h]
fffff800`028c7fa2 488bbdc8000000 mov     rdi,qword ptr [rbp+0C8h]
fffff800`028c7fa9 488bb5d0000000 mov     rsi,qword ptr [rbp+0D0h]
fffff800`028c7fb0 654c8b1c2588010000 mov     r11,qword ptr gs:[188h]
fffff800`028c7fb9 f685f000000001 test    byte ptr [rbp+0F0h],1
fffff800`028c7fc0 0f844f010000 je      nt!KiSystemServiceExit+0x17a (fffff800`028c8115)
fffff800`028c7fc6 440f20c1 mov     rcx,cr8
```

Buscando en los names lo encontramos en IDA



Allí esta la llamada en ese call r10 y retornaría al INC ese.



Un poco más arriba están las tablas, y realiza las operaciones que vimos hasta que resuelve la entrada y salta en el CALL R10.

Por supuesto esto pertenece a ntoskrnl desde allí luego decide que tabla usar si su propia tabla o la de win32ksys.

```

1: kd> x nt!*KiSystemServiceRepeat
fffff800`028c7d00 nt!KiSystemServiceRepeat (KiSystemServiceRepeat)

```

¿Ese es el inicio de la función, podremos poner un breakpoint allí?

Arranco el proceso y switcheo el contexto cuando está detenido

```

PROCESS fffff800c5cc360
  SessionId: 1 Cid: 17b0 Peb: 7fffffd6000 ParentCid: 13c8
  DirBase: 5686c000 ObjectTable: fffff8a010c99f90 HandleCount: 14.
  Image: ConsoleApplication4.exe

1: kd> .process /i fffff800c5cc360
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
1: kd> g
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus:
fffff800`028c0b50 cc int 3
0: kd> !process -1 0
PROCESS fffff800c5cc360
  SessionId: 1 Cid: 17b0 Peb: 7fffffd6000 ParentCid: 13c8
  DirBase: 5686c000 ObjectTable: fffff8a010c99f90 HandleCount: 14.
  Image: ConsoleApplication4.exe

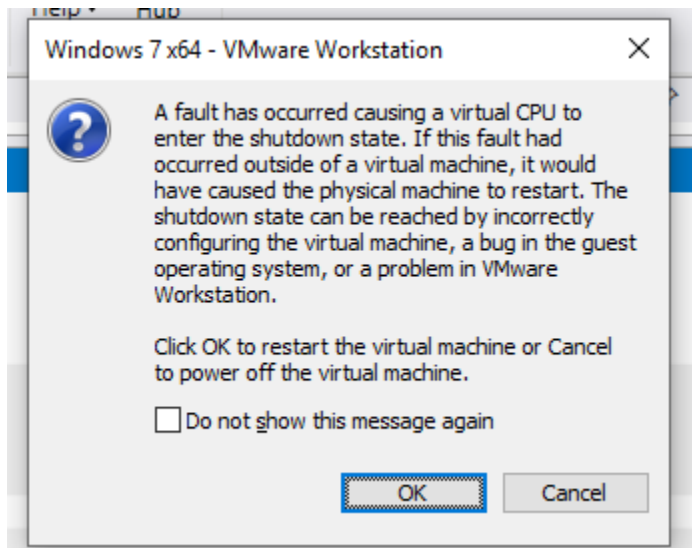
```

```
Image: ConsoleApplication4.exe

0: kd> x nt!*KiSystemCall64
fffffa800`028c7d00 nt!KiSystemCall64 (KiSystemCall64)

ba e1 /p fffffa800c5cc360 nt!KiSystemCall64
0: kd>
```

Espero que no se rompa todo veamos.



Glup por eso no deja entrar traceando desde SYSENTER jeje porque se rompe todo el sistema y realiza un BSOD.

Bueno de cualquier manera ya sabemos como hallar la función adonde salta desde user a kernel, tanto calculándola como usando el IDA, así que bueno, cuando necesitemos ya vimos que podemos hallarla y poner un breakpoint en la función donde salta y seguir desde allí, evidentemente parar antes detiene muchos procesos importante que la maquina no puede manejar y BSOD.

Pero logramos nuestro objetivo entender las tablas y saber calcular adonde saltara.

Hasta la próxima

Ricardo Narvaja

