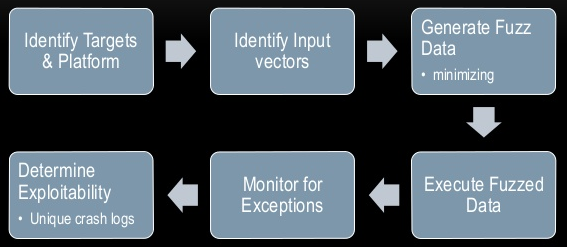
From Fuzzing to Exploit

##### 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>

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

**Binary Diffing:**

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 Diffing 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.

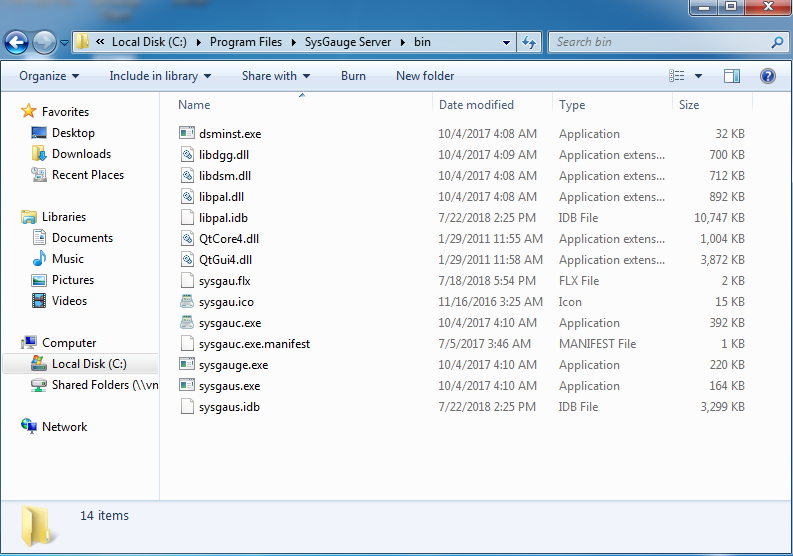
## **Comprender la comunicación y los inputs del programa vulnerable**

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

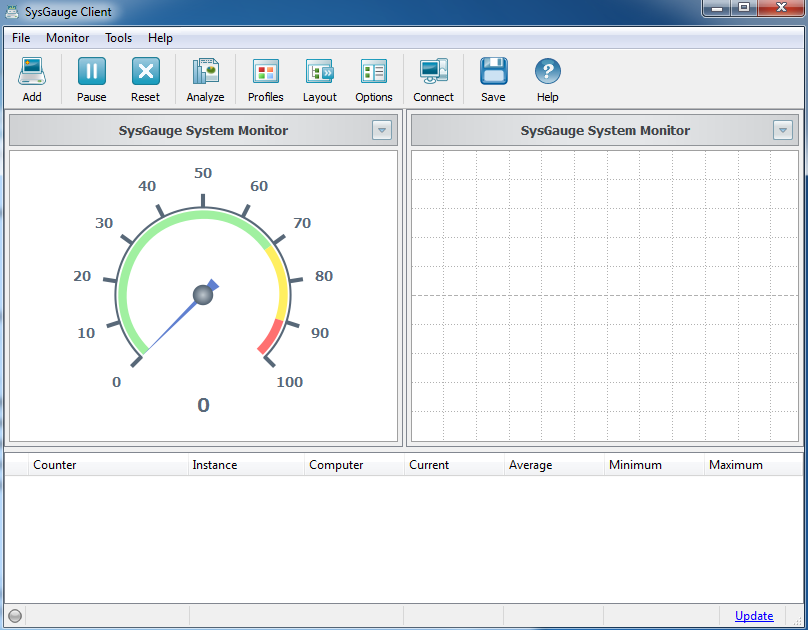


Luego de su instalación, podemos ver que los archivos del programa alojados en:

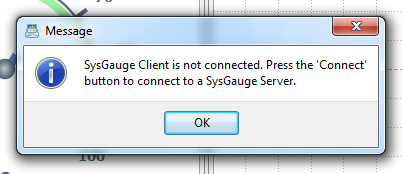
**C:\Program Files\SysGauge Server\bin**



Luego de esto iniciamos el programa por primera vez:

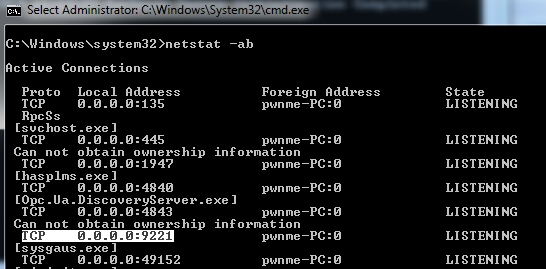


Al intentar realizar cualquier acción vemos que el cliente nos solicita conectarse a SysGause Server:



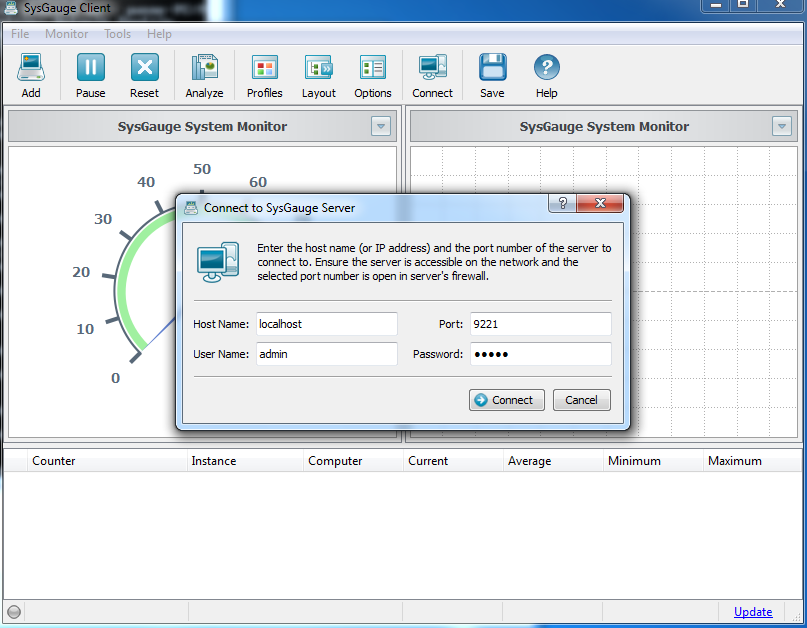
Según la descripción del CVE la vulnerabilidad se encuentra en el Servidor de SysGause que corre en el puerto 9221, por lo que listamos los procesos del Sistema con el comando: **netstat -ab**

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



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.

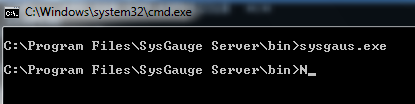
Sabiendo esto realizamos una conexión desde el cliente al servidor utilizando el botón connect.



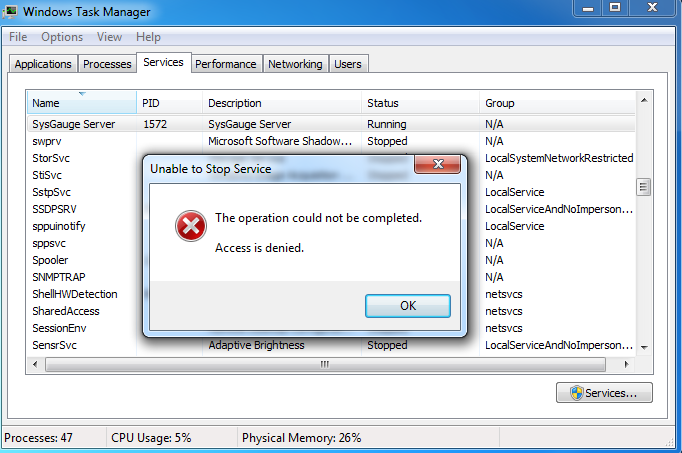
Luego de esto vemos como todo empieza a operar con normalidad:



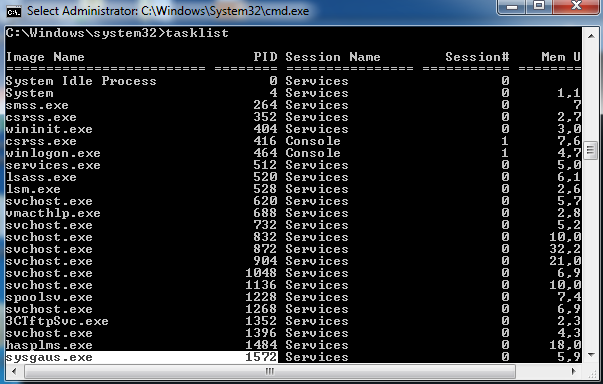
Sabiendo que el responsable de iniciar el Servidor del software es el binario **sysgaus.exe** debemos comprender como funciona y como iniciarlo ya que si lo ejecutamos directamente no sucede nada.

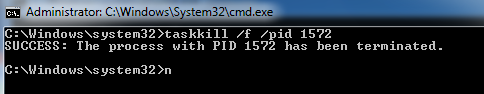


Además, al intentar matarlo vemos que corre como un servicio y con mayores privilegios que nuestro usuario actual:



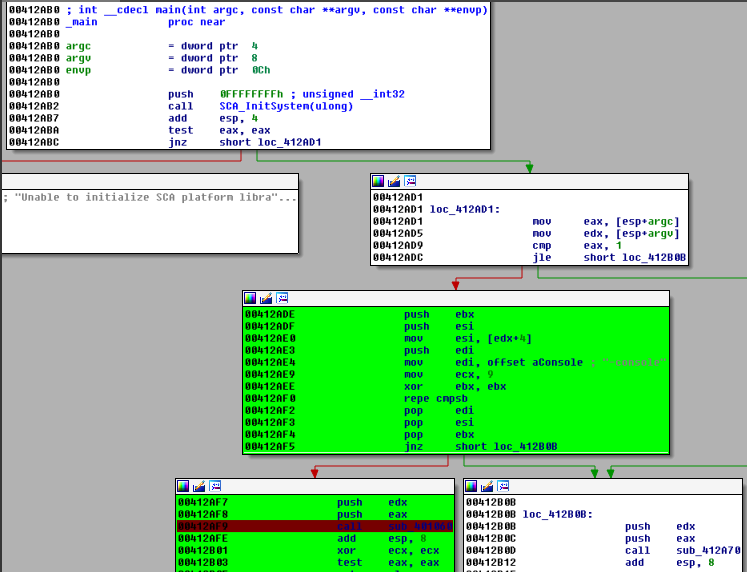
Por lo que debemos matar el proceso desde una terminal Administrativa utilizando el comando **tasklist** y **taskkill /f /pid {pidNumber}**





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

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

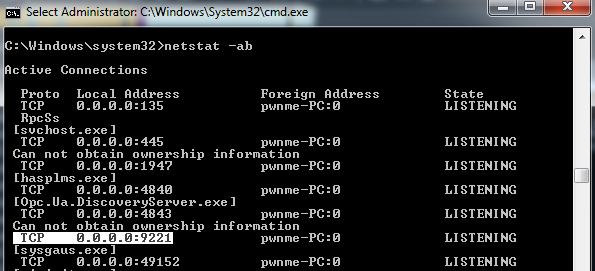


Donde podemos apreciar que el programa espera el parámetro **-console** para iniciar la ejecución.

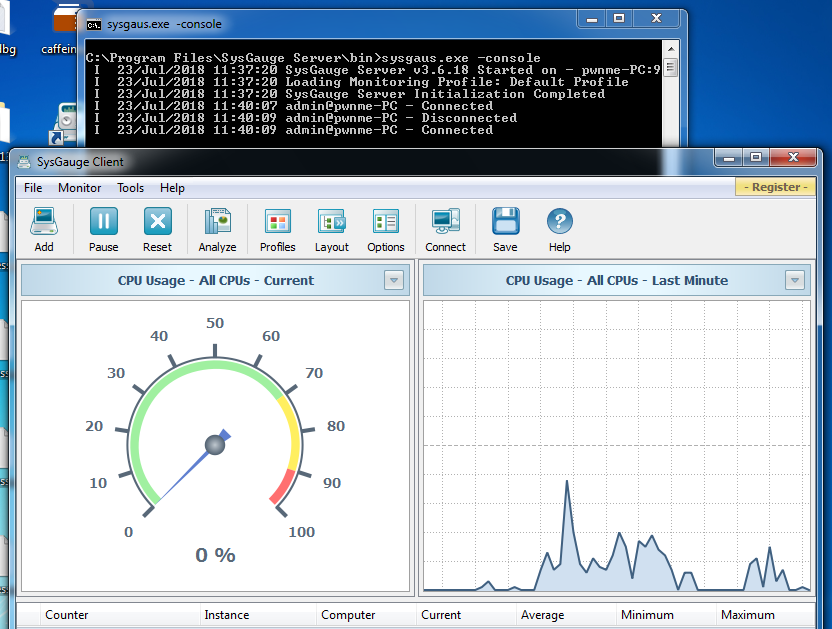
Probamos ejecutar el binario de la siguiente forma: **sysgaus.exe -console**



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



Para estar 100% seguros nos conectaremos con el cliente de SysGause 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.

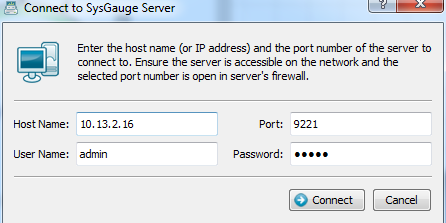
Para esto, en primera instancia simularemos ser el Servidor poniendo a la escucha el puerto 9221 en otra maquina utilizando netcat con el siguiente comando:

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

OSX Machine: **netcat -l 9221 -vv**



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:

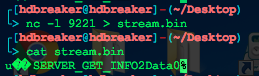


Obviamente, el paquete contendrá caracteres no printeables por lo que dumpereamos el stream de bytes a un archivo que luego podremos leer y convertir a hex-bytes. Para esto utilizamos el siguiente comando:

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

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

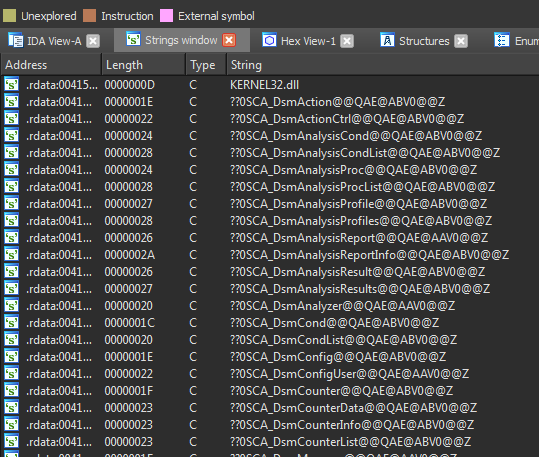
Y repetimos el proceso de conectar el cliente con nuestro Fake Server:



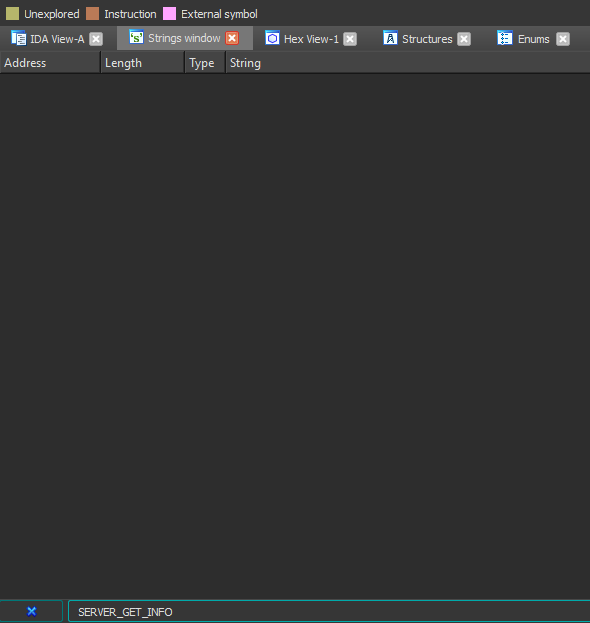
De esta forma somos capaces de capturar un primer mensaje valido, de una forma podremos analizarlo, convertirlo a hexadecimal y reproducirlo utilizando sockets en Python.

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.



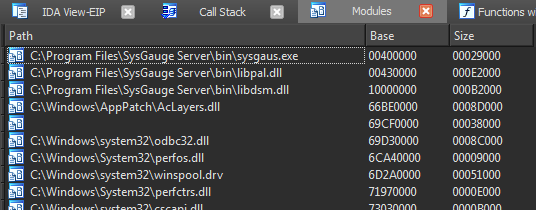
Luego con **CTRL + F** podemos filtrar los resultados buscando por **SERVER\_GET\_INFO**

****

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

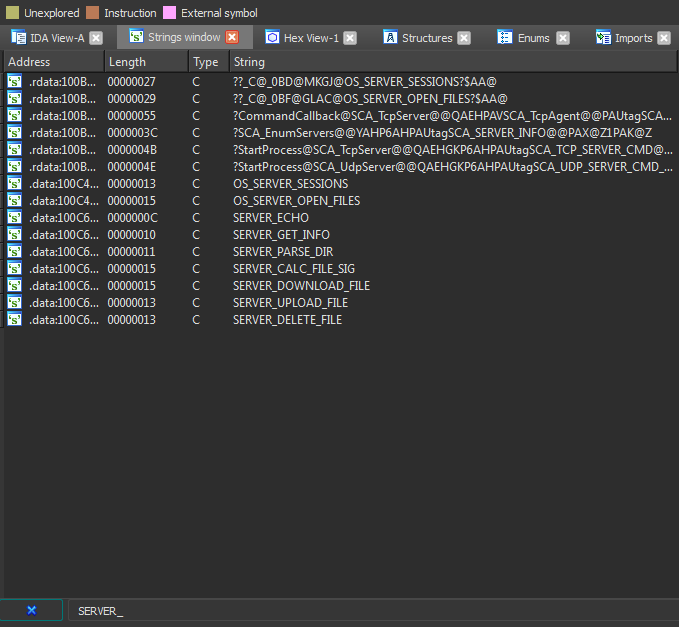
Para averiguarlo, configuramos el debugger como **Local Win32 debugger** y presionamos la tecla **F9** para iniciar la ejecución.

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.



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:



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:

Posible lista de comandos:

**. SERVER\_ECHO**

**. SERVER\_GET\_INFO**

**. SERVER\_PARSE\_DIR**

**. SERVER\_CALC\_FILE\_SIG**

**. SERVER\_DOWNLOAD\_FILE**

**. SERVER\_UPLOAD\_FILE**

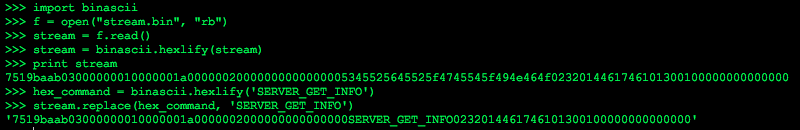
**. SERVER\_DELETE\_FILE**

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:

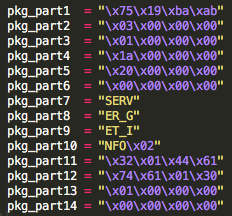
**\x75\x19\xba\xab\x03\x00\x00\x00\x01\x00\x00\x00\x1a\x00\x00\x00\x20\x00\x00\x00\x00\x00\x00\x00SERVER\_GET\_INFO\x02\x32\x01\x44\x61\x74\x61\x01\x30\x01\x00\x00\x00\x00\x00\x00\x00**

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

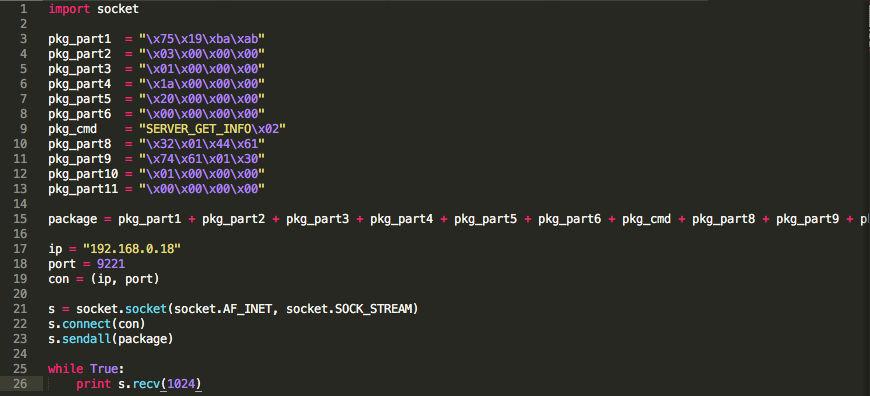


Sabiendo que este el servidor esta basado en arquitectura x86 es probable que esta información sea procesada de 4 bytes. Esto suena razonable ya que el largo del stream es divisible por 4: **56/4 = 14** por lo que no rompería la alineación del Stack.

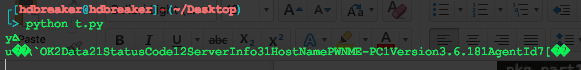
Comenzamos a programar nuestro cliente en Python bajo esta suposición:



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:



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/>

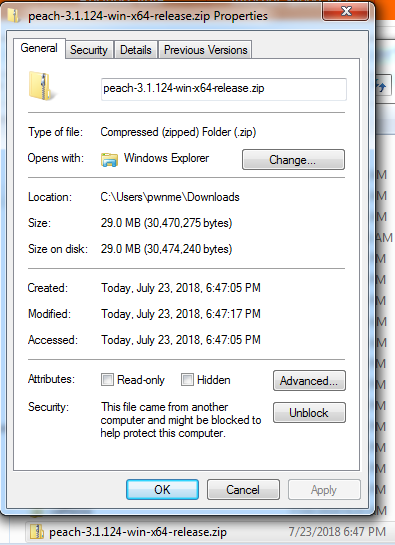
Peach, permite crear, utilizando XML, archivos que le indican como debe fuzzearse un paquete.

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:

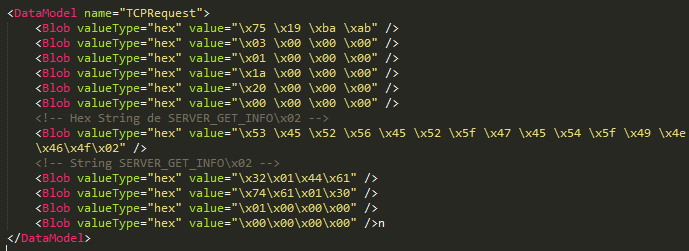
**. DataModel:** Es donde definiremos la estructura de nuestro paquete.

**. StateModel:** Indica como o bajo que protocolo se comunica el cliente con el servidor.

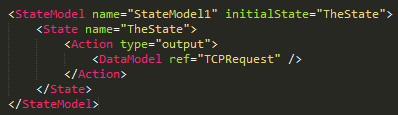
**. Agent:** Contiene la definición del Agente que utiliza Peach para comunicar su proceso de fuzzing con el Servidor a fuzzear, en el se define la ruta al programa a fuzzear y también se encarga de controlar el estado del Servidor, si este crashea lo vuelve a iniciar.

**. Test:** Define que Agent y que StateModel se utilizaran en el proceso de fuzzing, donde se encuentra a la escucha el Servidor a ser Fuzzeado y donde se almacenaran los logs cuando el Server crashee.

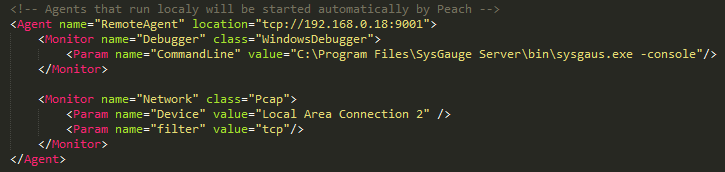
Primero definiremos nuestro **DataModel** de a 4 bytes de tipo blob y un campo string de nuestro pkg\_cmd (definido en hexadecimal ya que requiere el carácter no printeable \x02) como hicimos en nuestro archivo Python quedando de la siguiente forma:



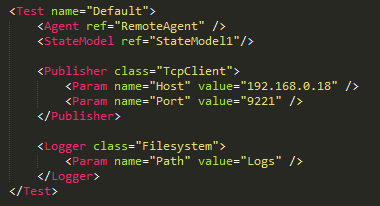
Definiremos nuestro **StateModel** como **TCPRequest**



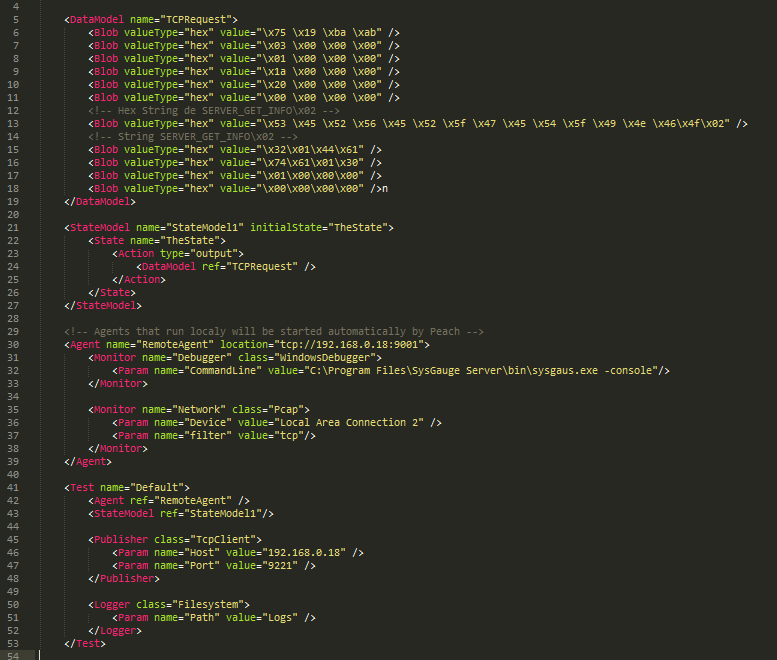
Nuestro **Agent** debe contener la **IP donde se ejecutará Peach** en este caso es la misma que la maquina local ya que no estamos utilizando Peach de forma remota. Además, indicaremos la **ruta del binario vulnerable y los argumentos para ejecutarlo** de forma correcta. Por último, también debemos indicar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



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).

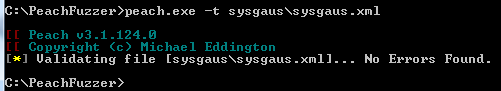


Uniendo todas las partes, nuestro XML con las reglas de Fuzzing para Peach se vería de la siguiente forma:

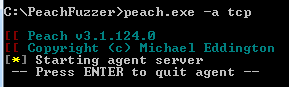


Guardaremos este archivo con el nombre **sysgaus.xml** en alguna carpeta creada dentro del directorio donde tenemos instalado Peach, en mi caso: **C:\PeachFuzzer\sysgaus**

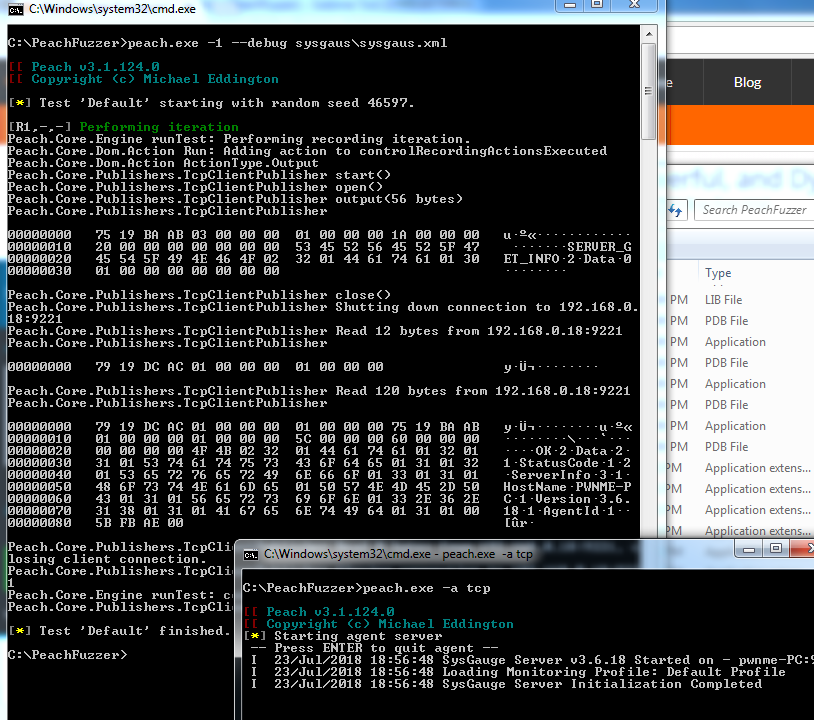
Antes de iniciar el proceso de fuzzing debemos verificar que la estructura de nuestro XML sea valida para esto iremos a la carpeta donde instalamos Peach Fuzzer, abriremos una terminal cmd y ejecutaremos el comando: **peach.exe -t sysgaus\sysgaus.xml**



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**



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**

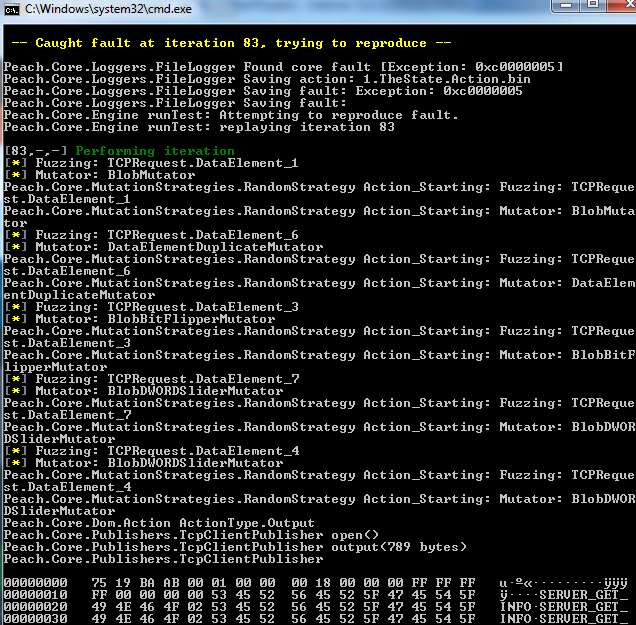


En la imagen podemos apreciar como el Peach Agent inicio de forma correcta el Servidor vulnerable. Y como Peach Fuzzer envió el primer paquete de testing sin fuzzear al Servidor y obtuvo como resultado una respuesta valida del Servidor.

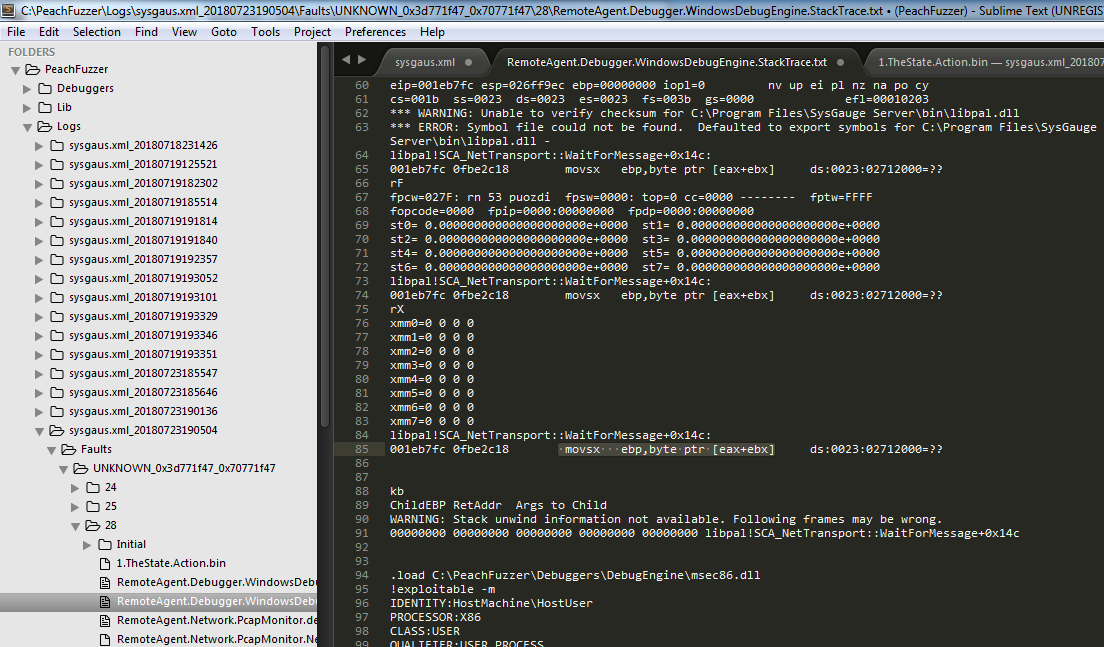
Con esto ya estamos listos para iniciar nuestro proceso de fuzzing, solo debemos dejar corriendo el Agente de Peach, y ejecutar en siguiente comando en otra terminal:

**peach.exe --debug sysgaus\sysgaus.exe**

Luego de un tiempo de Fuzzing encontraremos nuestro primer crash:

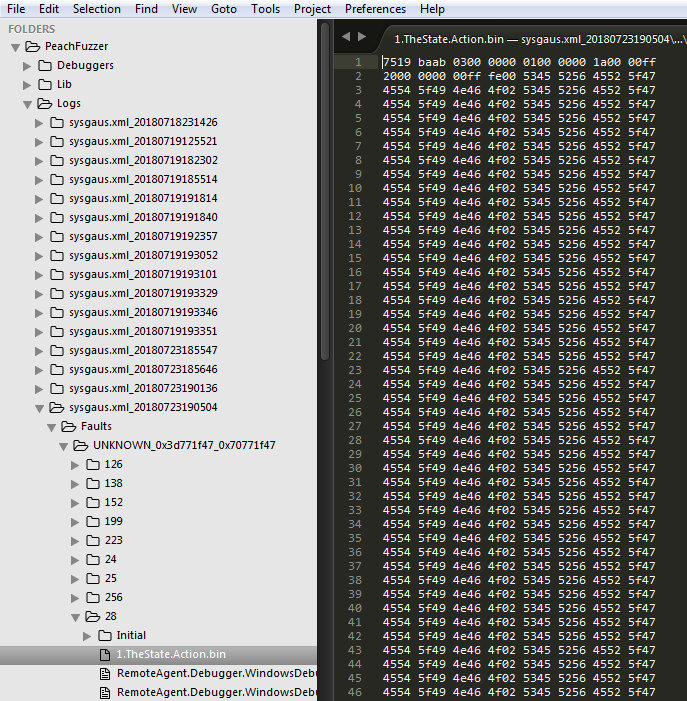


Si Peach Fuzzer logra reproducir el crash guardara un logfile con el estado de memoria en el momento del crash y un archivo con el paquete para reproducirlo en la carpeta: **C:\PeachFuzzer\Logs\sysgaus.xml\_{timestamp}**



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]**

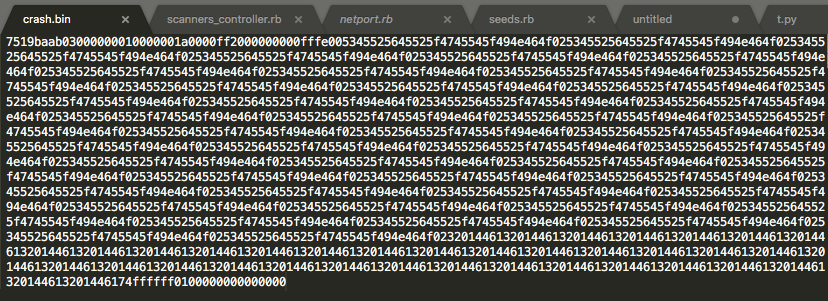
Además, el archivo **TheState.Action.bin** contiene el mensaje binario que se envió al Servidor para hacerlo crashear.



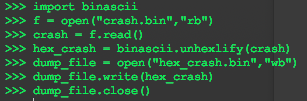
Este archivo será nuestro punto de partida para reproducir el crash en el debugger y analizar el bug.

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.

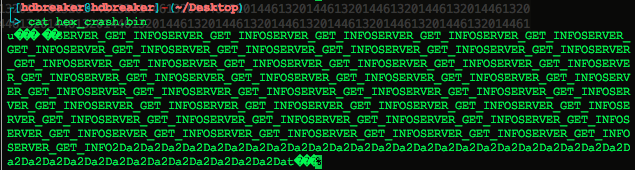
Primero copiaremos los bytes a un archivo y eliminaremos los espacios y saltos de línea:



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

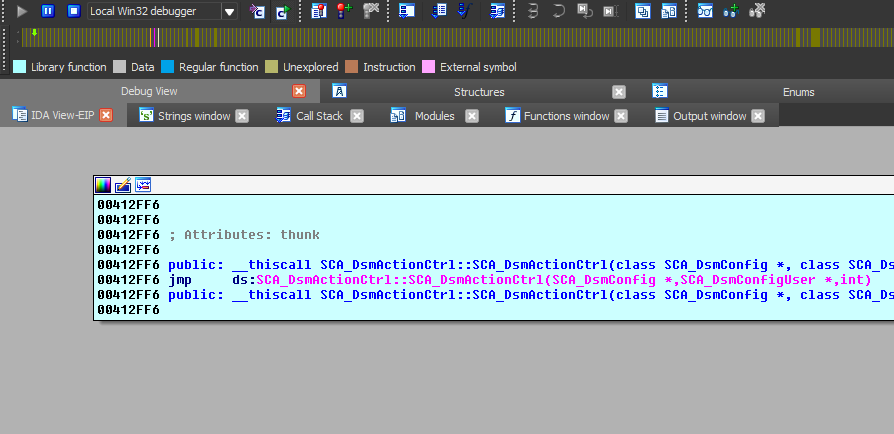


Al ver el contenido del archivo **hex\_crash.bin** podemos ver un Stream de datos validos que podemos enviarle al Servidor:



## **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.

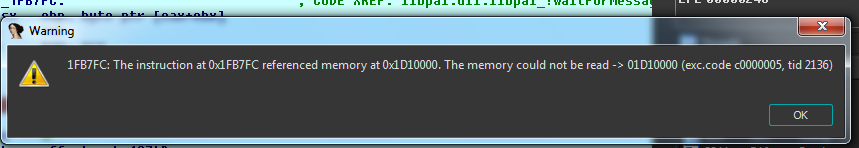


Y enviaremos el contenido de hex\_crash.bin al servidor de la siguiente forma:

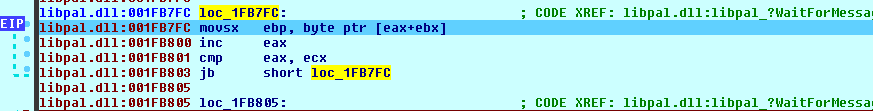
**cat hex\_crash.bin | nc 192.168.0.18 9221**

****

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

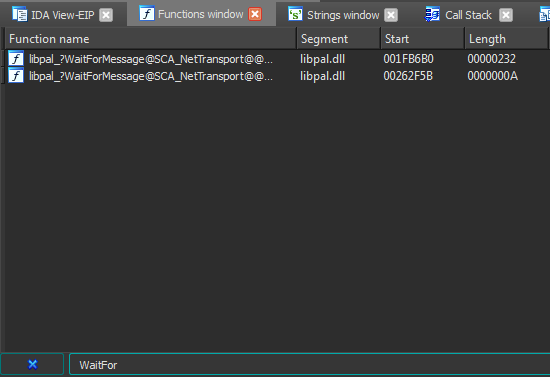


Podemos apreciar que el binario crashea en la misma instrucción que nos mostraba el log de Peach:



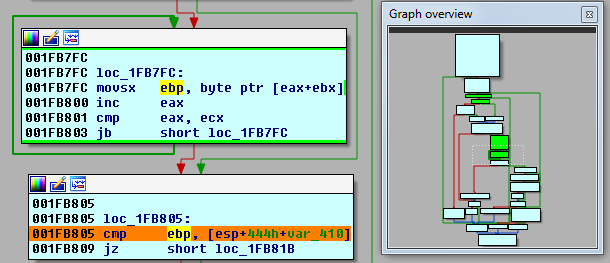
Y también podemos notar que el crash se produce en la función **WaitForMessage** de la librería **libpal.dll**

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**)



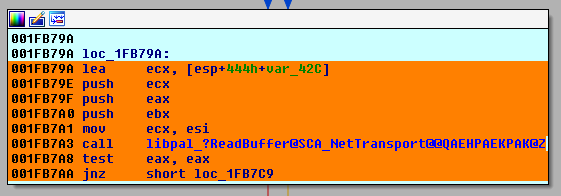
Damos doble clic sobre la primera función y presionamos **la barra espaciadora** esto nos permitirá ver la función utilizando el **Graph View** de IDA y nos permitirá analizarla mejor.

Ubicamos la sección del crash:

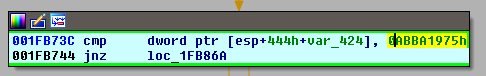


Y comenzamos a analizar, que sucede antes del crash y los alrededores de la función.

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:



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

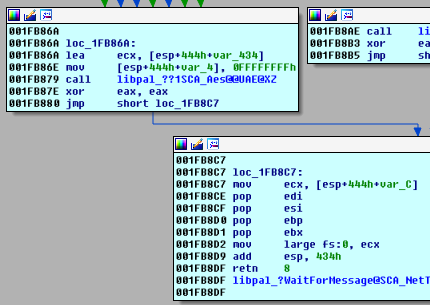


Siendo **ABBA1975** el Little Endian de **"\x75\x19\xba\xab"** el cual es la primera parte de nuestro paquete:



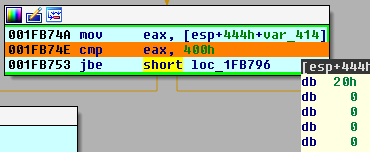
Con esta información podemos deducir que nos encontramos ante el parser del binario.

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.

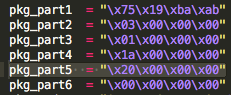


Por lo que deducimos que los primeros 4 bytes del paquete son el **header** del mismo e indica como debe ser tratado el mensaje.

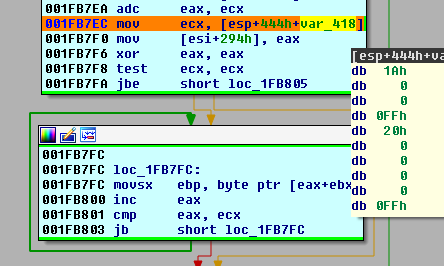
Debajo de la comparación con el header se encuentra una segunda comparación:



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.

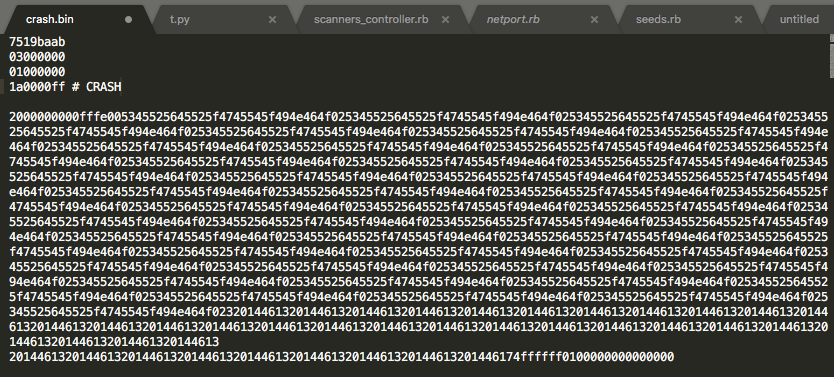


Si continuamos analizando el flujo de ejecución llegamos a otra instrucción controlada por nuestro mensaje:

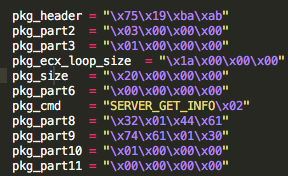


Vemos que var\_418 contiene “1A0000FF” luego es movido a ecx, y ecx es utilizado para controlar el flujo del bucle donde se produce el crash. Es probable que este loop sea donde se leen byte a byte el mensaje que enviamos.

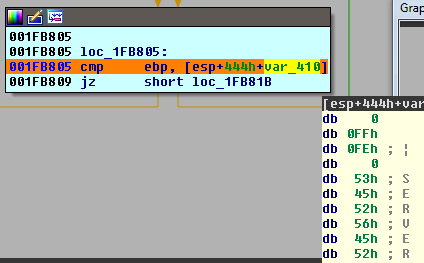
Si analizamos el mensaje que enviamos para generar el crash podemos determinar que **var\_418** se encuentra bajo nuestro control:



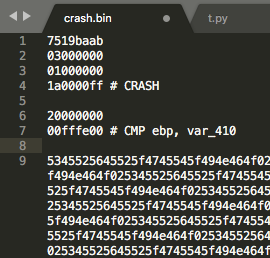
Esto nos permite ir reverseando el protocolo en base a los registros y variables que controlamos:



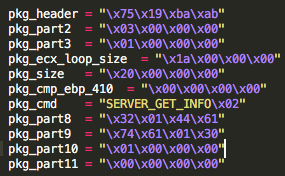
Luego del loop podemos ver la siguiente comparación:



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



Vamos modificando nuestro script con esta información:



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**



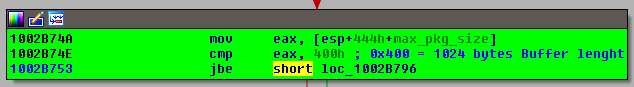
Podemos suponer que la función **DecryptBuffer** esta esperando el mensaje cifrado en AES con una key que no conocemos, por lo que idealmente deberíamos evitar pasar por esa sección.

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:

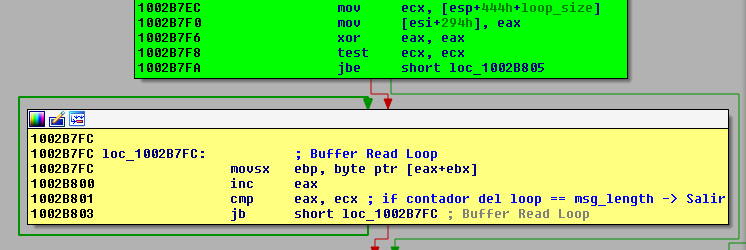
**Detectamos el Header del paquete:**



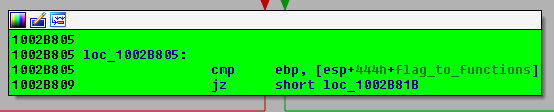
**Detectamos el max\_pkg\_size:**

****

**Detectamos el loop size del bufferReader:**

****

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

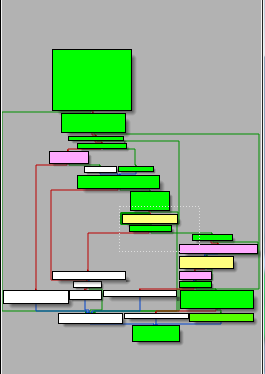
****

**Siendo el flujo ideal de un paquete valido el siguiente:**

**Verde:** Flujo de ejecución ideal

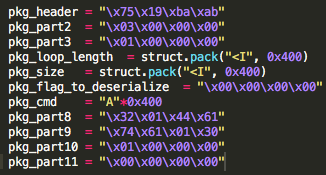
**Rosa:** Basic Blocks a Evitar

**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.

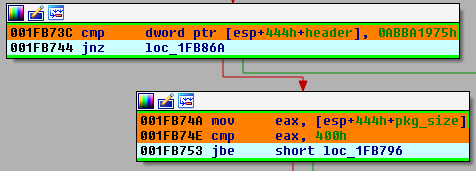
Nuestro script quedaría de la siguiente forma:



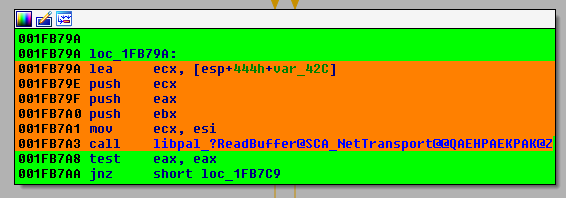
Especulamos que **pkg\_loop\_length** y **pkg\_size** están relacionados de alguna forma entre si por ello los coloco con el mismo size para evitar el crash en loop. Además, podemos pensar que **pkg\_cmd** debe ser el buffer donde espera los **1024 bytes**.

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

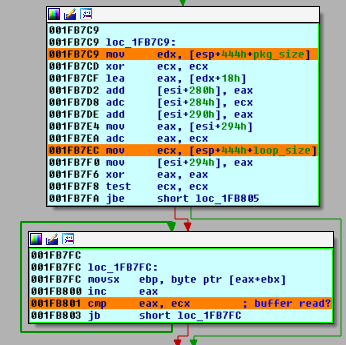
**Package Header y Package Size:**



**Buffer Read:**



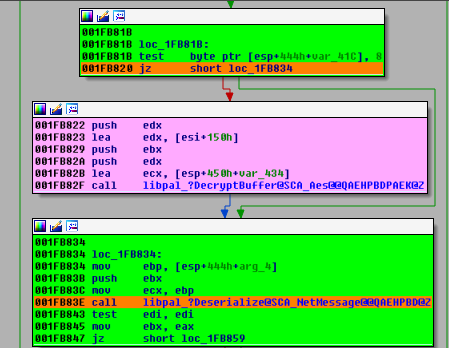
**Loop Crash:**



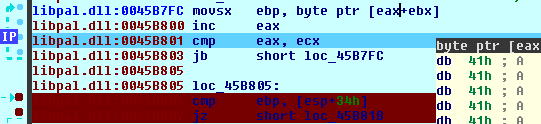
**Flag que salta hacia las funciones:**



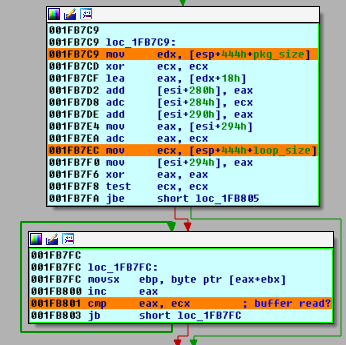
**Ultimas funciones en juego:**



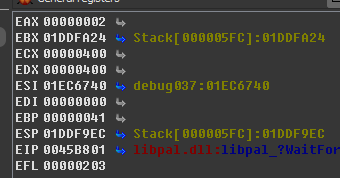
Ejecutamos el script en Python y analizamos el proceso en IDA, al llegar al loop donde especulamos que leía nuestro buffer sucede lo siguiente:



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



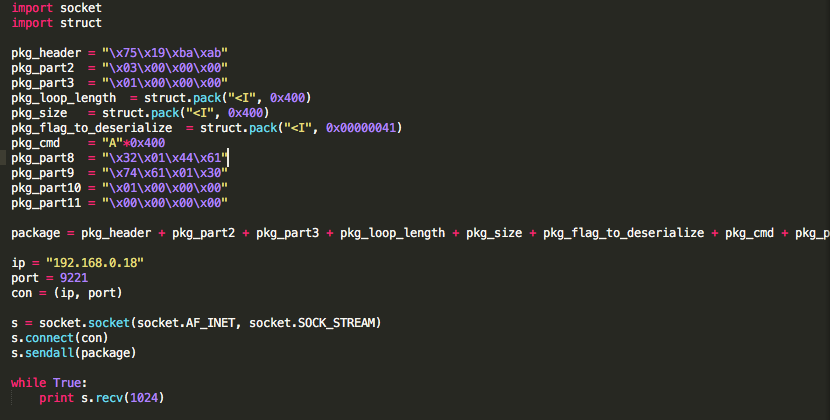
Podemos ver como efectivamente controlamos loop\_size y pkg\_size en los registros:



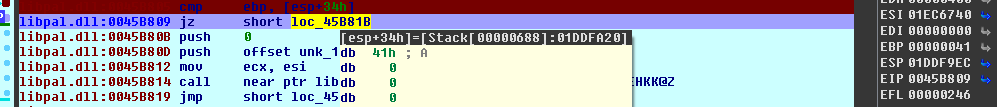
Al llegar a la comparación de **flag\_to\_functions**, podemos ver como la comparación espera el valor 0x41 (Seguro relacionado a nuestro buffer) en ebp, y nosotros estamos enviando solo ceros:



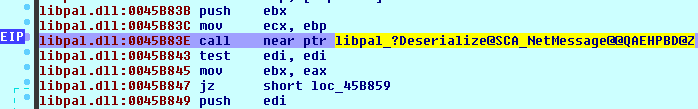
Esto nos redireccionara hacia el final de la función, por lo que realizamos una rápida corrección en nuestro archivo y ejecutamos nuevamente el script.



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

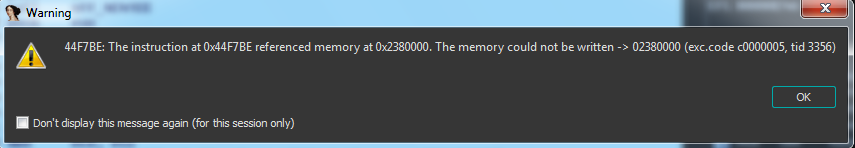


Al continuar un flujo normal de ejecución podemos apreciar como evitamos sin hacer nada la función **DecryptBuffer** y llegamos a la función **Deserialize** cumpliendo de esta forma con el flujo esperado:

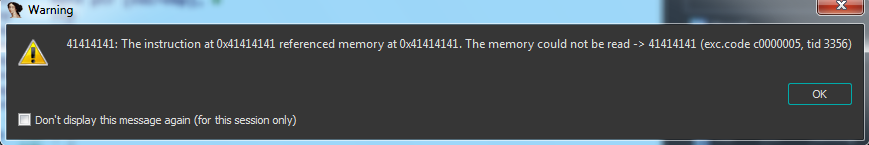


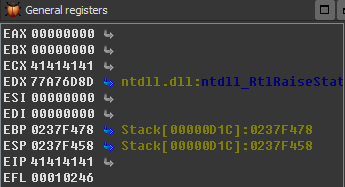
En este punto hemos evitado el crash, tenemos múltiples registros y variables bajo nuestro control y un paquete valido con un size máximo de 1024 bytes lleno de nuestras “A”.

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:



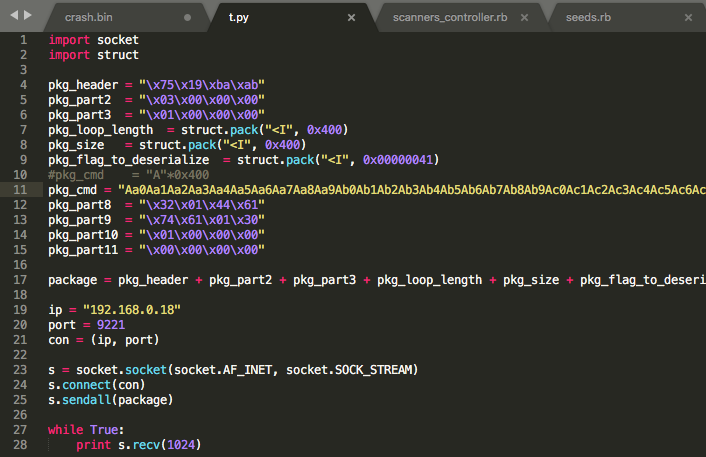


## **Explotación del Binario**

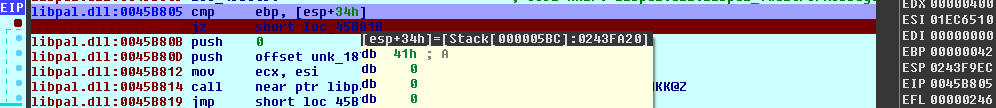
Ahora solo nos falta detectar cuales son los bytes con los que pisamos EIP, para esto vamos a generar un string pattern de 1024 bytes utilizando la ayuda de Metasploit.



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



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:



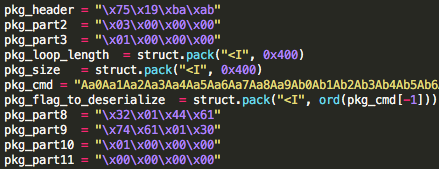
Podemos ver como EBP tiene el valor **0x42** y que esta comparando con **0x41**

Si analizamos nuestro script con detenimiento:

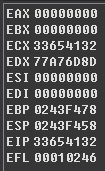


Podemos determinar que **[esp+34] (flag\_to\_functions)** es controlado por **pkg\_flag\_to\_deserialize** y que **0x42** de **ebp** esta siendo controlado por el ultimo byte de **pkg\_cmd** (Note esto luego de varios intentos)

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

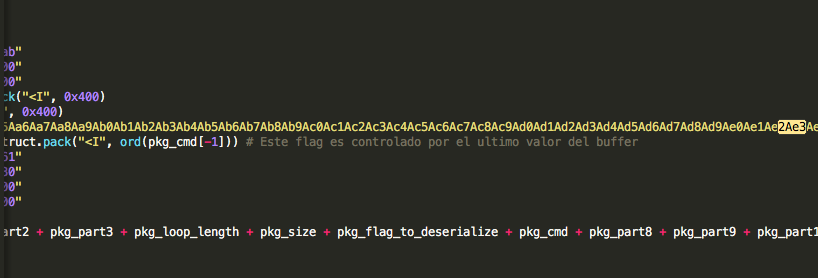


Al realizar una nueva ejecución y analizar con IDA, obtenemos que EIP apunta a **0x33654132**:

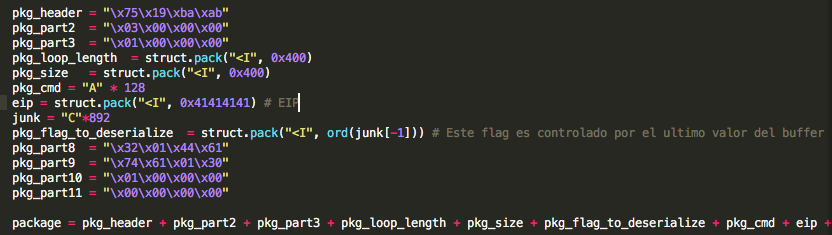


Esta dirección es equivalente al string Little Endian **3eA2**, que en nuestro mensaje se vería como **2Ae3**

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

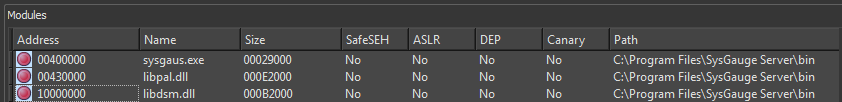


Ajustamos nuestro script para reflejar el control de EIP:



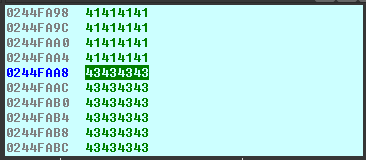
Es importante notar la cantidad de bytes en **pkg\_cmd** (**128 bytes)** y en **junk (892 bytes)** por lo que, de ser posible, seria mas cómodo trabajar nuestro shellcode dentro de los **892 bytes de junk.**

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



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**

****

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

****

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)**

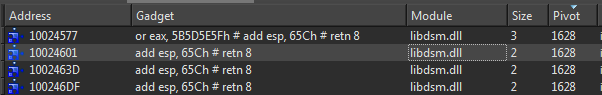
Pasar la ejecución al stack para ejecutar nuestro buffer (No DEP)

. **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 \*\***

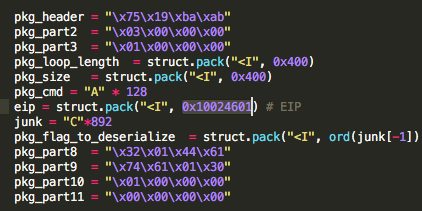
**Buscando un Stack Pivot Gadget**



Verificamos que el pívot exista en 0x10024601:



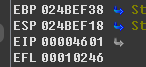
Sustituimos EIP con la dirección del Stack Pivot: **0x10024601** en nuestro exploit.



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



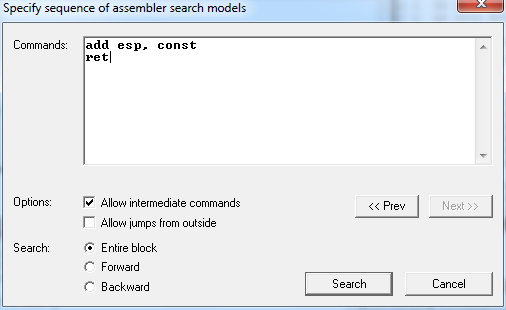
Y ejecutaremos el exploit, pero vemos que EIP no refleja la dirección de EIP que enviamos:



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**

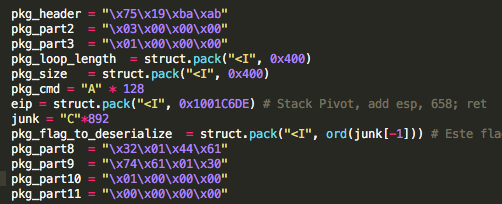


Y comenzamos a buscar algún gadget que cumpla la condición de hacer un **add esp, 0x650** o mayor y que el address no contenga el badchar **“\x02”**

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



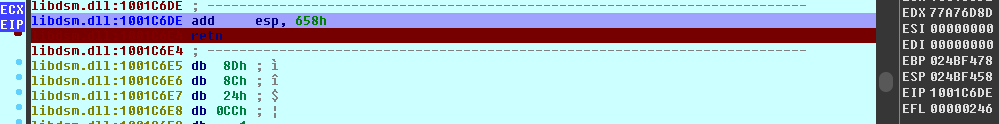
Sustituimos EIP en el exploit por esta dirección: **0x1001C6DE**

****

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

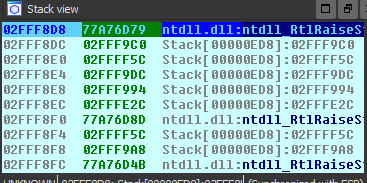


Y lanzamos el exploit una vez mas:

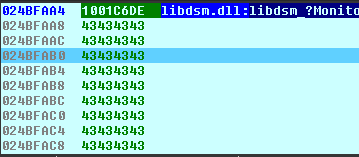


Al momento del crash EIP saltara hacia el stack pivot. El cual moverá el Stack hacia nuestro buffer:

**Stack antes del Pivot:**



**Stack luego del Pivot:**

****

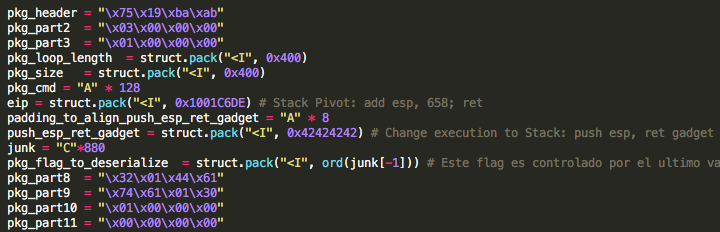
En este punto la ejecución del programa se encuentra detenida en el siguiente **RET**:



El cual obtendrá la dirección de EIP desde el stack controlado por nuestro buffer, hacienda 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:



Es importante aclarar que se ajusto el largo de junk:

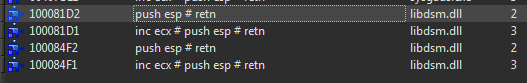
**Junk length antes: 892**

**Junk length después: 880**

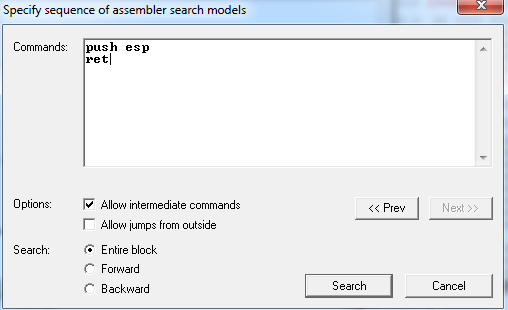
Esto debido a que agregamos 12 bytes al exploit entre el **padding para alinear el segundo gadget y el gadget** **push esp, ret** que estaban siendo ocupados por “C” en nuestro buffer.

## **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.

****

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



Modificamos nuestro exploit para reflejar el gadget:



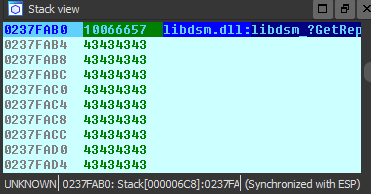
Ejecutamos el binario con IDA y colocamos un breakpoint en **0x10066657**



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



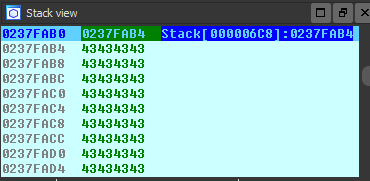
Veamos el Stack en este momento:



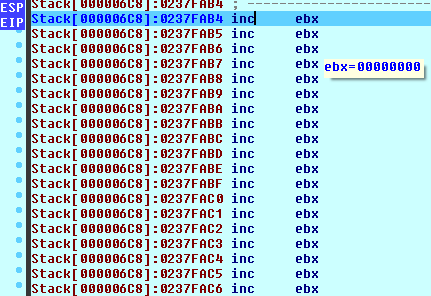
Vemos como **luego del Stack Pivot**, **RET** tomara el valor del Segundo gadget (**push esp, ret**) del Stack por lo que saltara hacia el.



Observemos el Stack en este momento:



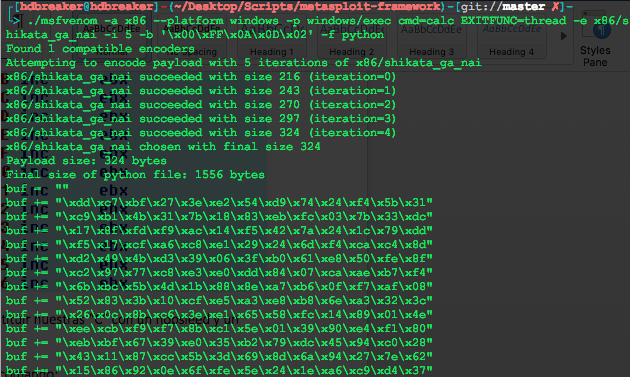
El **RET** del ultimo gadget pasara la ejecución del programa directamente hacia nuestras “C”



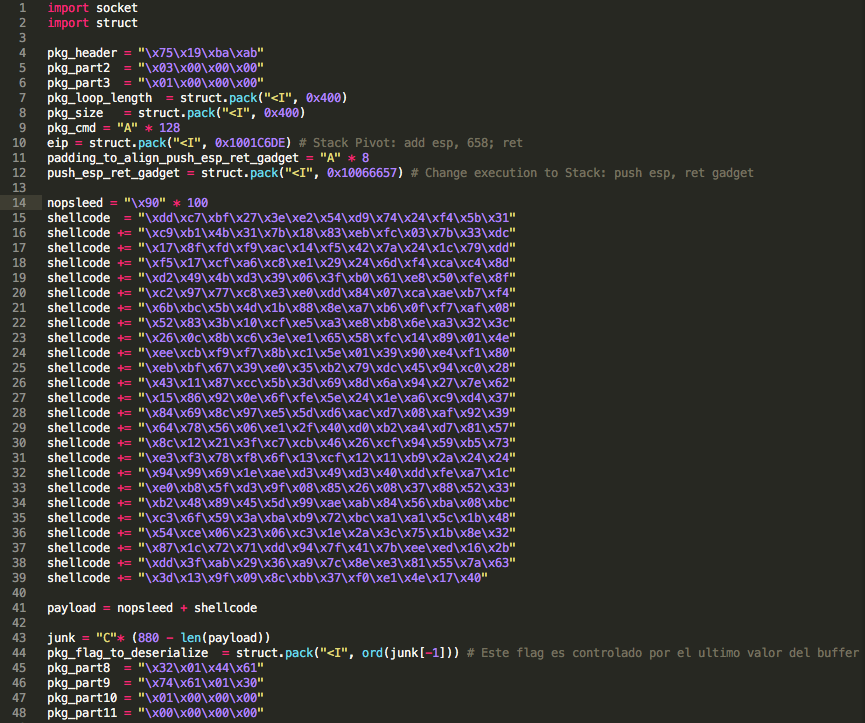
Para completar la explotación, bastaría con sustituir nuestras “C” con un nopsleed y un shellcode.

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**

****

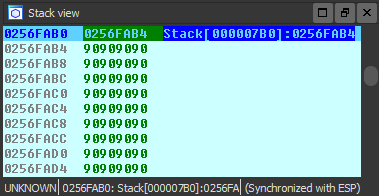
Modificamos nuestro exploit para agregar un nopsleed de 100 bytes, agregamos nuestro shellcode de 324 bytes y ajustamos el junk de “C” quedando el exploit de la siguiente esta forma:



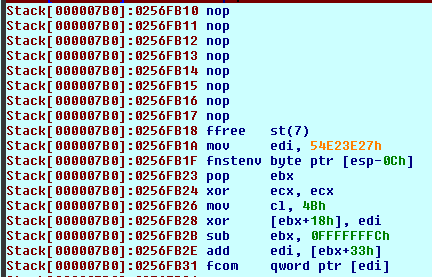
Ejecutamos el programa con IDA, lanzamos el exploit y seguimos la ejecución sobre los breakpoints hasta llegar al segundo gadget:



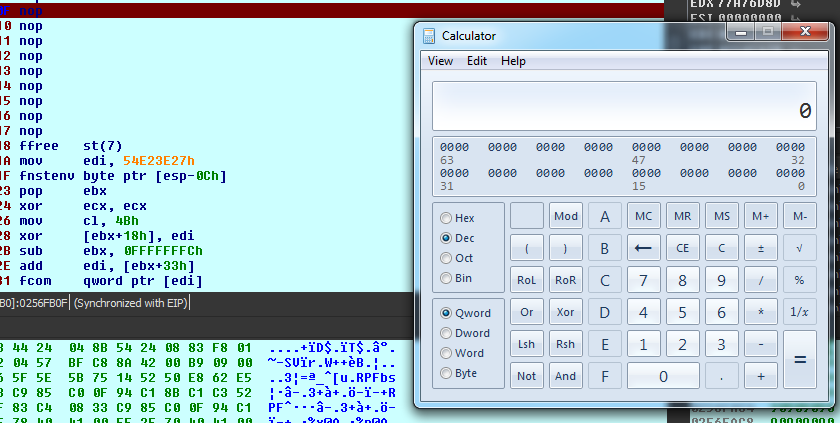
Observemos el Stack en este momento:



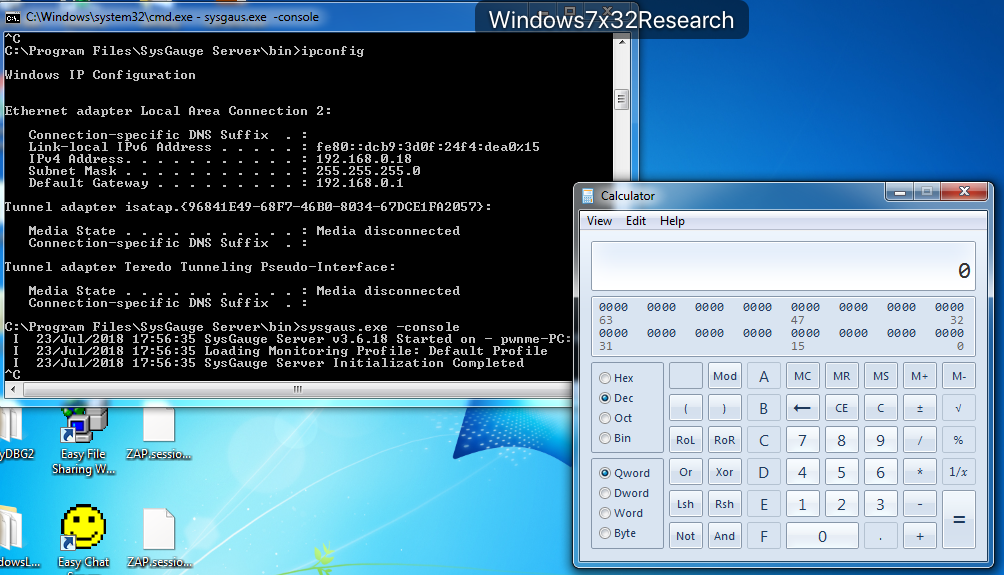
Vemos como el **RET** del ultimo gadget saltara hacia el Stack justo donde esta situado nuestro nopsleed.



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.

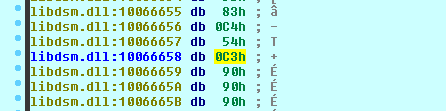
Con esto doy por concluido este tutorial, es el primero que realizo para la comunidad de **CSL\_Exploit**, espero sea de utilidad para todos!

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.

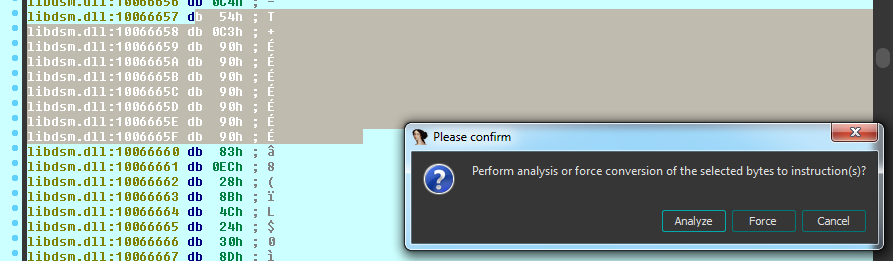


**TIP de Utilidad:**

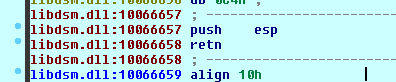
**Si al saltar hacia una dirección de una librería que fue cargada en memoria vemos algo como esto:**

****

**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:**

****