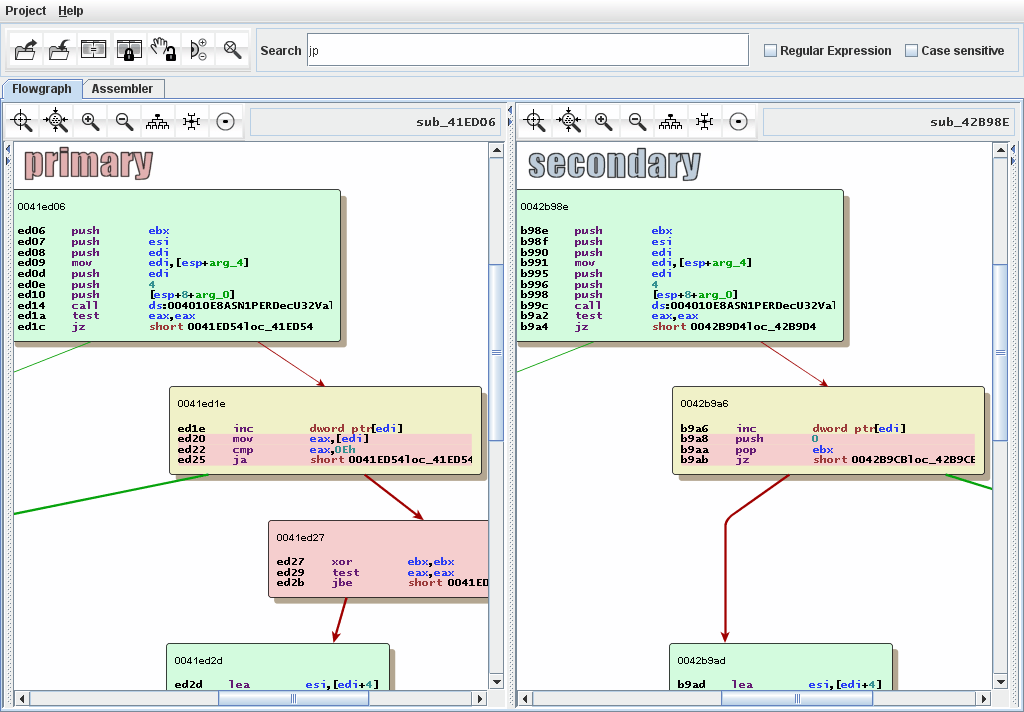
**Binary Diffing a Structured Binary**

##### SEIG Modbus Driver v3.34

CVE-2013-0662



**By hdbreaker**

**CVE Description:**

The Modbus Serial Driver creates a listener on Port 27700/TCP. When a connection is made, the Modbus Application Header is first read into a buffer. If a large buffer size is specified in this header, a stack-based buffer overflow results.

**Link:** <https://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=27505>

**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!

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

<https://github.com/hdbreaker/Ricnar-Exploit-Solutions/tree/master/Medium/CVE-2013-0662-SEIG-Modbus-Driver-v3.34/VERSION%203.4>

Y su versión fixeada desde:

<https://github.com/hdbreaker/Ricnar-Exploit-Solutions/tree/master/Medium/CVE-2013-0662-SEIG-Modbus-Driver-v3.34/VERSION%203.5>

**Binary Diffing (La técnica Elegida):**

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.

Los pasos a seguir son los siguientes:

1. Instalar el plugin BinDiff de IDA Pro
2. Instalar la versión fixeada y detectar el binario que escucha en el **puerto 27700**
3. Realizar una copia del binario en una carpeta correctamente nombrada
4. Eliminar la versión fixeada
5. Instalar versión vulnerable
6. Realizar una copia del binario vulnerable en una carpeta correctamente nombrada
7. Desensamblar con IDA Pro la versión corregida con el fin de generar el archivo **idb** correspondiente
8. Desensamblar con IDA Pro la versión vulnerable del programa.
9. Utilizar BinDiff para encontrar las diferencias entre las versiones del software

**Es importante remarcar que la vulnerabilidad es explotable en Windows XP, ya que si se instala el software en Windows 7 o posterior el binario que se encarga de manejar las conexiones al puerto 27700 no es el vulnerable. El environment elegido es un Windows XP x86 SP3.**

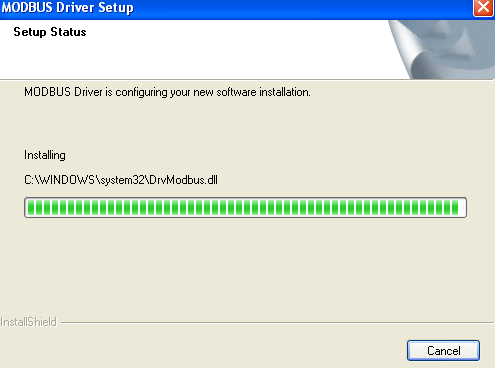
# **Instalar el plugin BinDiff de IDA Pro**

Los pasos para la instalación del plugin pueden descargase en el siguiente link: <https://www.zynamics.com/software.html>

La instalación es completamente automatizada, al completarla IDA Pro será capaz de realizar Binary Diffing utilizando la herramienta BinDiff de Zynamics.

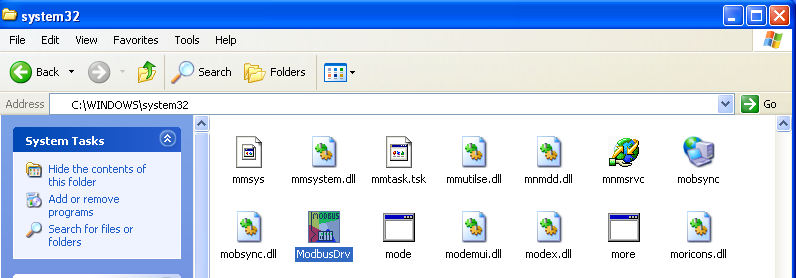
# **Instalar la versión fixeada y detectar el binario que escucha en el puerto 27700**

Instalamos la versión corregida del software:

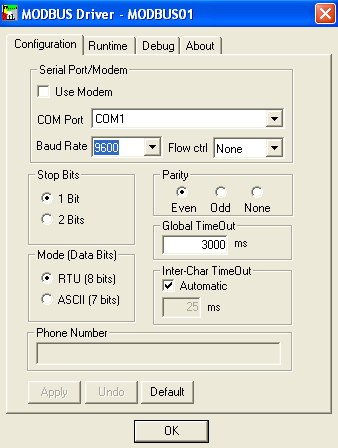


Luego de su instalación, podemos ver los archivos del programa alojados en:

**C:\WINDOWS\system32**

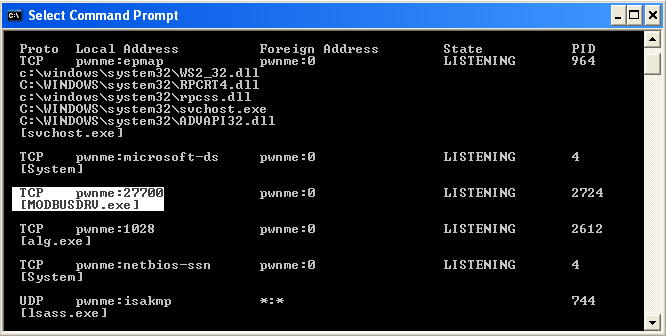


Luego de esto iniciamos el programa por primera vez:



Según la descripción del CVE la vulnerabilidad se encuentra en el Servicio que escucha en el **puerto 27700**, por lo que listamos los puertos en escucha del Sistema con el comando: **netstat -ab**

Para esto necesitamos una terminal cmd:

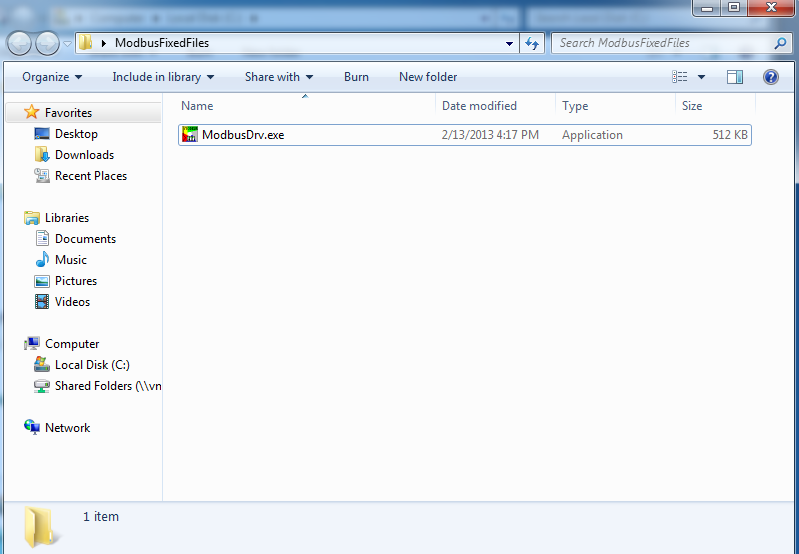


Podemos observar que el programa responsable de escuchar en ese puerto es el binario **MODBUSDRV.exe** y este corre en 0.0.0.0 (todas las interfaces de red) en el **puerto 27700**.

Sabiendo esto realizamos una copia del programa corregido a una carpeta correctamente nombrada.

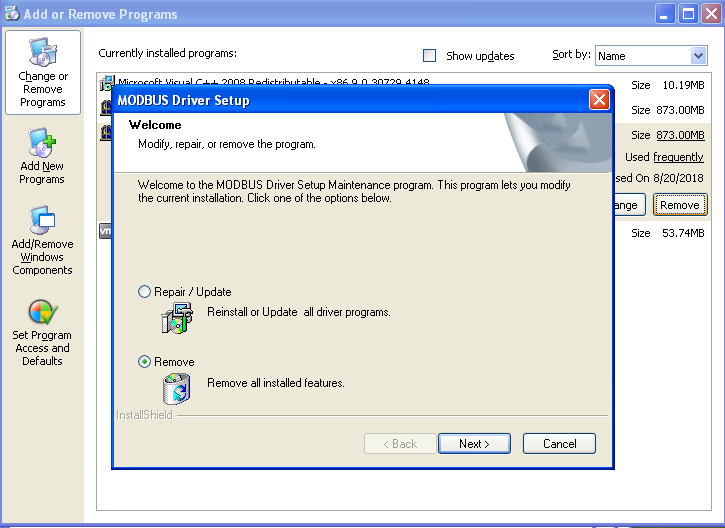
# **Realizar una copia del binario en una carpeta correctamente nombrada.**

**Nuestra maquina de análisis es un Windows 7 x86 por lo que copiamos los archivos a una carpeta creada en este sistema operativo.**



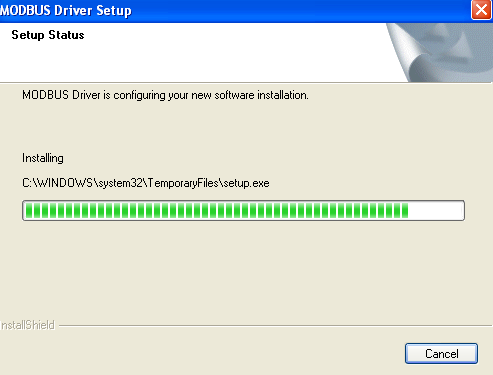
# **Eliminar la versión fixeada**

Es importante asegurarse de este proceso ya que si el bug se encuentra en alguna librería del software y esta no es eliminada en el proceso de desinstalación, podría interferir con el proceso de reversing.



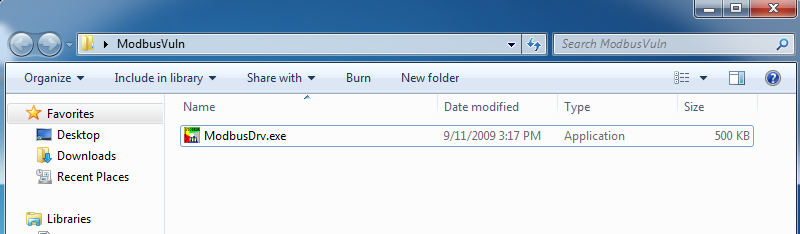
# **Instalar versión vulnerable**

Realizamos la instalación de la versión vulnerable (**Modbus Driver Suite v3.4**).



# **Realizar una copia del binario vulnerable en una carpeta correctamente nombrada.**

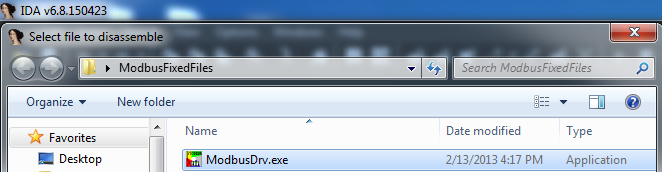
**Nuestra maquina de análisis es un Windows 7 x86 por lo que copiamos los archivos en una carpeta del sistema de análisis.**



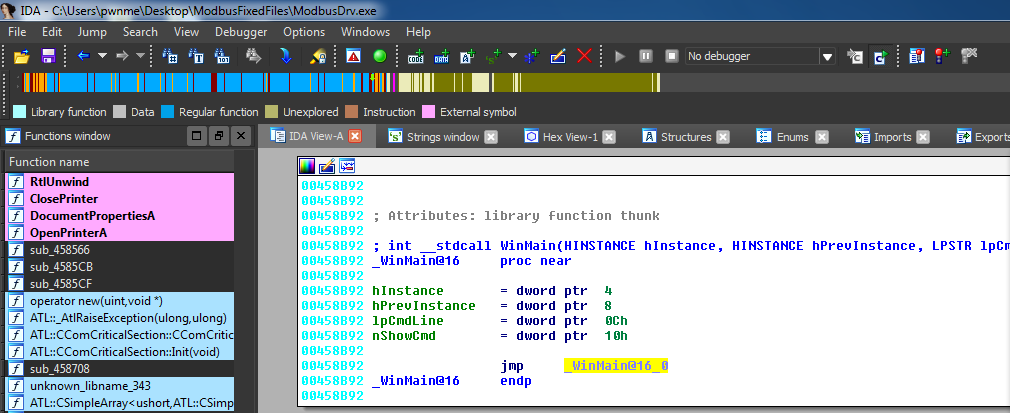
# **Desensamblar con IDA Pro la versión corregida con el fin de generar el archivo idb correspondiente**

Abrimos la versión fixeada del software con IDA Pro, esperamos que complete el análisis y cerramos el proyecto para generar el archivo idb.

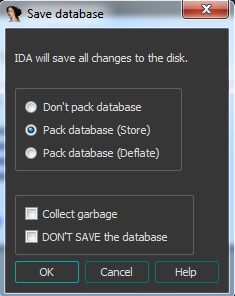
Abrimos el Binario con IDA:



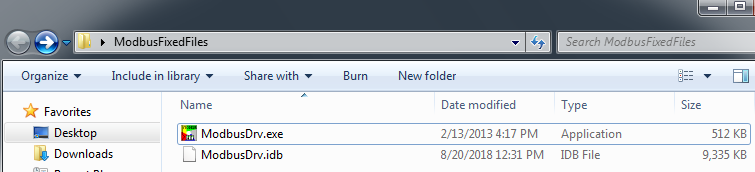
Esperamos que termine el análisis:

****

Cerramos el binario marcando la opción Pack Database:



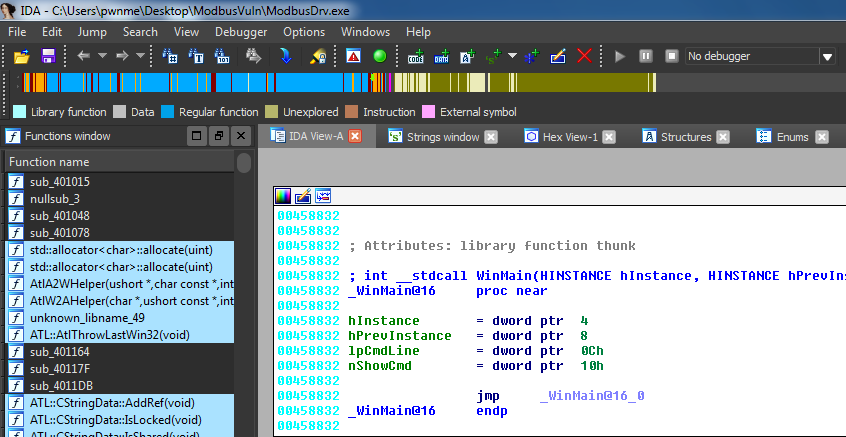
Luego de este proceso podemos ver cómo se generó el archivo **idb** con toda la información relacionada a la versión vulnerable del programa.



## Este archivo es muy importante ya que lo utilizaremos para realizar el **BinDiff contra la versión vulnerable**.

# **Desensamblar con IDA Pro la versión vulnerable del programa**

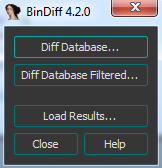
Abrimos la versión vulnerable del software con IDA Pro y esperamos que complete el análisis.



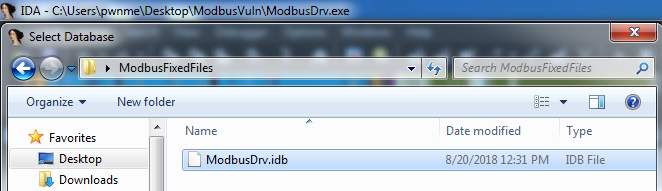
Una vez completado el proceso podemos utilizar el plugin de BinDiff para encontrar las diferencias entre la versión vulnerable y la versión corregida.

# **Utilizar BinDiff para encontrar las diferencias entre las versiones del software**

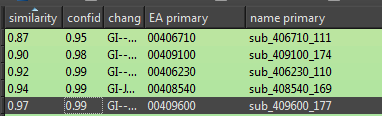
BinDiff puede ser accedido en la siguiente ruta: **Edit -> Plugins -> BinDiff 4.2.0** o presionando **Ctrl + 6.**



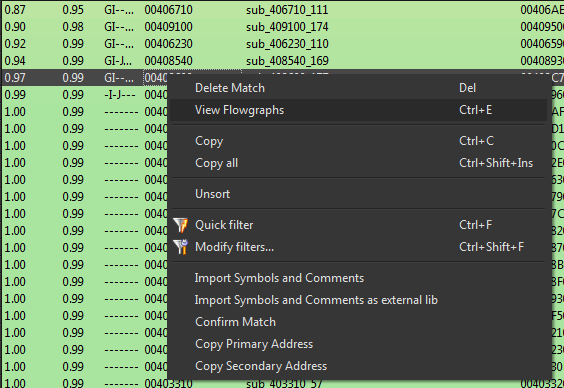
Presionamos **Diff Database** y seleccionamos el archivo **idb** de **la versión corregida del software**.



## Esperamos algunos segundos y obtenemos, entre otra información, el siguiente listado donde se muestran las funciones que poseen diferencias entre la versión vulnerable y la versión corregida.



## Esto reduce la superficie de research a solo 5 funciones que deben ser analizadas una a una para detectar el bug, para esto nos posicionamos sobre una función y presionamos **Ctrl + E** (botón derecho -> View Flow)

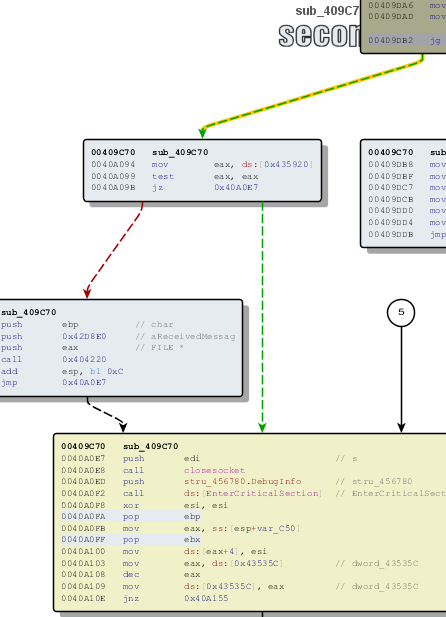


Avanzando entre ellas hay una que llama la atención (**sub\_409600**) donde se puede ver lo siguiente:

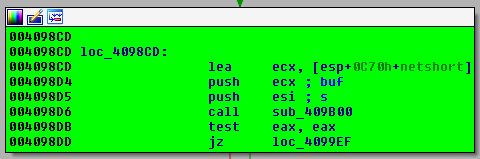


Podemos observar lo que podría ser un posible parche para evitar un buffer overflow, podemos ver cómo múltiples funciones de red se ven involucradas en el parche y posteriormente en el address: **00409DA0** se compara **ebp** con **0x40E**

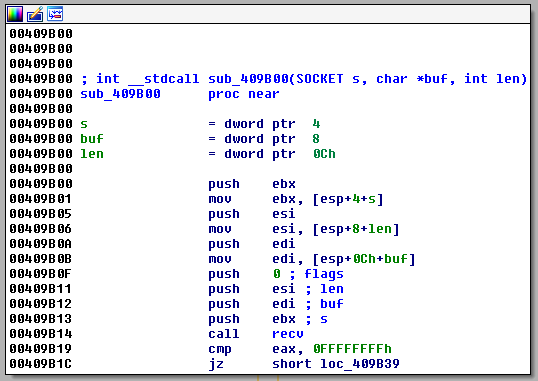
En la versión parcheada, **si ebp es mayor que 0x40E** el flujo se dirige a un **socket close** que terminaría la conexión del cliente como puede verse a continuación:



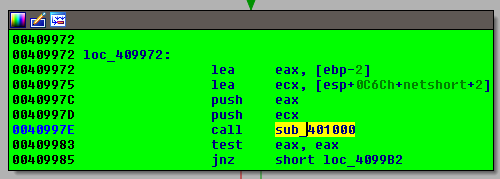
Pero ¿qué sucede en la versión vulnerable? Si analizamos el flujo con IDA Pro podemos ver que más abajo se realiza el siguiente llamado a la función (**sub\_409B00**):



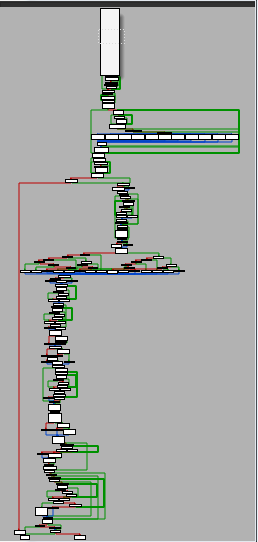
Al analizar el llamado podemos ver que se trata de un **recv:**



En un caso ideal el flujo de ejecución continuaría hasta llegar a la siguiente llamada (**sub\_401000**):

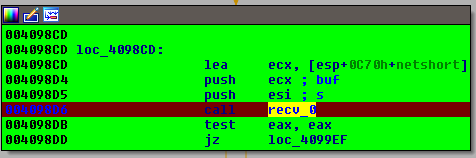
****

Al analizar esta función podemos ver que se trata de una función muy grande probablemente relacionada a un parser:

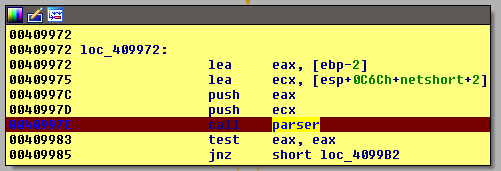


El mismo contiene múltiples llamados a **strcpy** y **memcpy** por lo que parece un buen punto de inicio, así que volvemos al principio y comenzamos a renombrar funciones quedando de la siguiente forma:

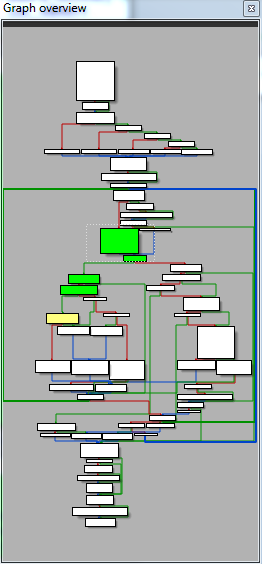
**Function Recv:**



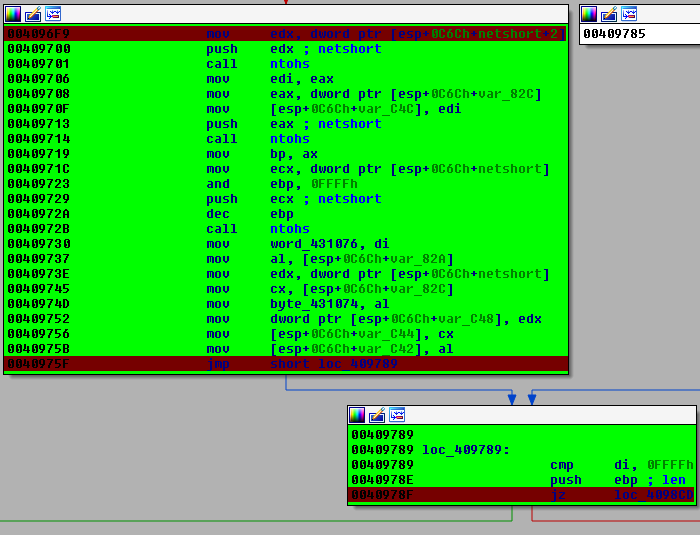
**Function Parser:**



**Graph Overview del flujo ideal:**

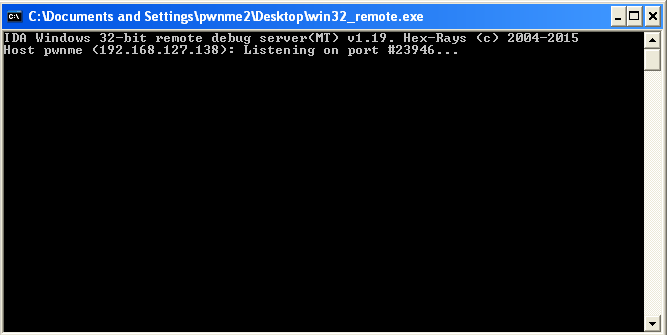
****

## Colocamos algunos breakpoints en el primer Basic block y nos encontramos listos para continuar.

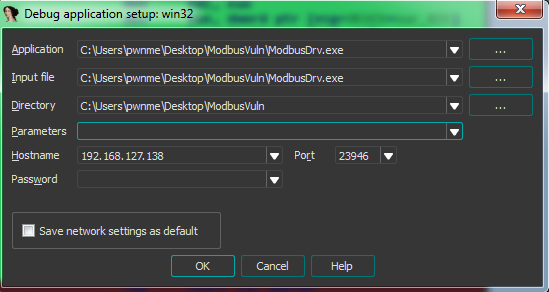


Una vez seteados los break points debemos configurar el debugger para trabajar de forma remota en nuestro **Windows XP SP3**,para esto utilizamos el servidor de debugging propio de IDA: **win32\_remote.exe**

Lo copiamos en la maquina virtual y lo ejecutamos:



Configuramos el debugger en ida de la siguiente forma:

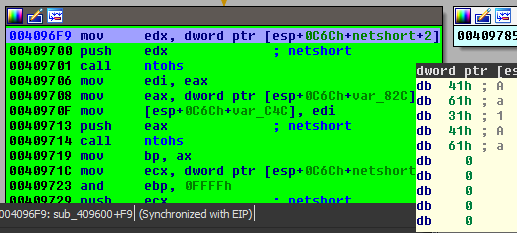


## Con el debugger listo, podemos attachearnos al programa y enviar nuestro primer mensaje.

## Para esto vamos a generar un string aleatorio de 40 Bytes y vamos a enviarlo por medio de Python sockets, esto para determinar si controlamos alguna variable o registro con el mensaje que enviamos. Nuestro script en Python se vería de la siguiente forma:



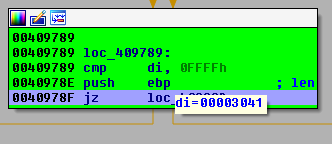
Ejecutamos el script y esperamos que se triggereen los breakpoints:



Podemos ver que lo primero que se evalúa es el string **Aa1Aa** que corresponden al string comprendido entre el byte 3 y 7 de nuestro message:

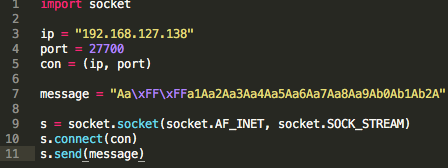


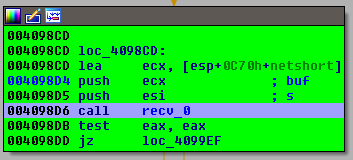
Al continuar la ejecución llegamos a la siguiente comparación:



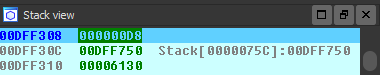
Podemos observar que el programa realiza una validación de lo que supondremos, es el message type (el cual nosotros controlamos con el valor 0x3041 que es igual a **0A – tercer y cuarto carácter de nuestro string**), en caso de que el **message type** sea **0xFFFF** este salta hacia la zona que nosotros queremos dirigirnos.

Modificamos estos bytes para cumplir con la condición:

  
  
Al volver a ejecutar el script la condición se cumple y nos envía hacia el **recv\_0**.

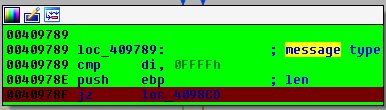


Al analizar el stack podemos ver los parámetros que son enviados como argumentos de esta función:

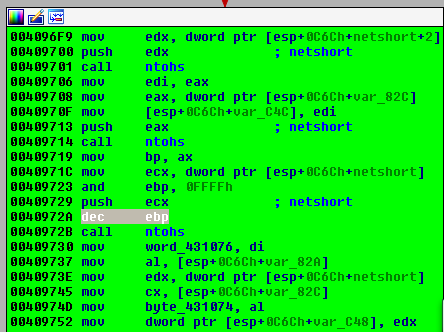


El primer valor es el socket, el segundo es la dirección del buffer donde la información será alojada y el tercer valor es el size a ser leído.

Este valor es pusheado al stack por la instrucción **push ebp** en el Basic block donde se realiza la comparación con el **message type**:



El registro **ebp** es seteado por el primer Basic block que analizamos, donde se le resta uno como puede apreciarse en la siguiente imagen:



Por lo que podemos determinar que el valor de **ebp** es en realidad **0x6131**, que es equivalente al string **a1**, valor que es controlado en nuestro string.

Al continuar la ejecución la función **recv\_0** falla al no poder leer un mensaje del tamaño solicitado y entra a un bucle esperando completar el buffer con próximos paquetes.

Teniendo en cuenta esta información podemos determinar que la función **recv\_0** recibe un mensaje que incluye como parte de su cuerpo el size del buffer a leer.

Con toda esta información podemos comprender que **el header del mensaje** esta compuesto por **7 bytes** de los cuales:

. Dos bytes no modifican el comportamiento del mensaje, por lo que los nombraremos como **padding bytes.**

. Dos bytes definen el **message\_type.**

. Dos bytes definen el **buffer\_size** a leer.

. Un byte que no interfiere con el mensaje, pero delimita el header por lo que lo llamaremos **header\_end**.

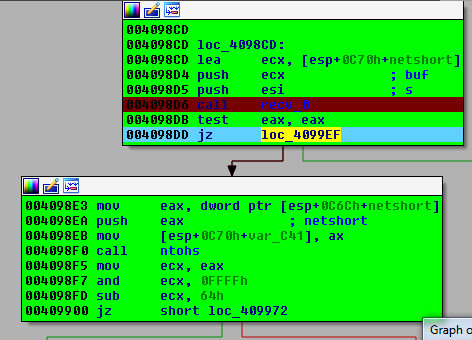
Teniendo en cuenta el parche:



Cualquier mensaje con un size mayor a **0x40E** debería desbordar el buffer, por lo que modificaremos nuestro script de la siguiente forma para intentar triggerear el bug:

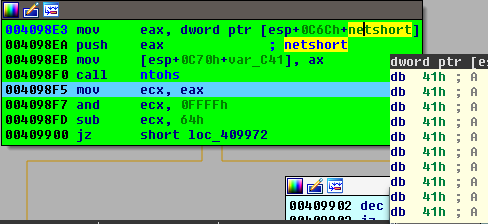


Al ejecutarlo podemos observar como la función **recv\_0** se completa con éxito y el programa sigue con su ejecución:



Luego del segundo Basic block se encuentra la llamada a la función **parser** donde probablemente el overflow se produzca, pero antes de llegar a ella se produce una comparación que desvía la ejecución del programa.

Esto es producto de **la resta de ecx con 0x64**, si observamos con detenimiento **ecx**, obtiene su valor de un **mov ecx, eax** y **eax** obtiene su valor desde **[esp+offset]**. Nos posicionamos sobre la estructura para visualizar el valor que contiene:



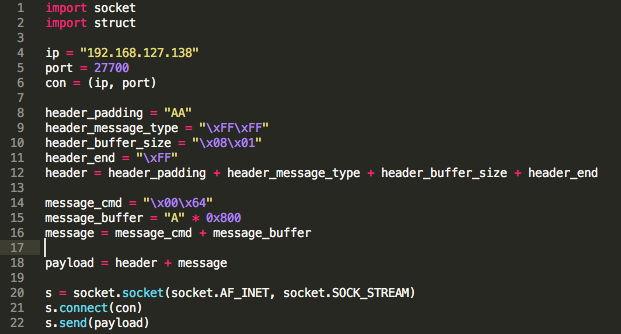
La instrucción **mov eax, [ebp+offset]** setea a **eax** con el valor de **0x41414141** el cual corresponde a los 4 primeros bytes del buffer enviado. Luego este valor es procesado por la función **ntohs** la cual convierte un short integer (2 bytes) de **Big Endian a Little Endian** y retorna el primer byte. Por ej.:

Si enviamos **\x41\x42** **Big Endian** la función **ntohs** hará un switch de estos valores a **\x42\x41** y retornará.

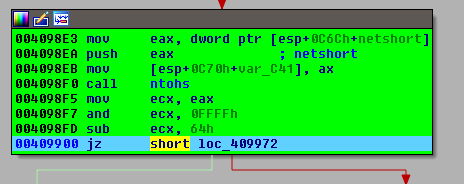
El resultado de **ntohs** es movido dentro de **ecx** el cual es luego restado contra **0x64**. Si la resta da como resultado 0, el programa continua su flujo hacia la función **parser.**

Entendido esto, podemos determinar que los primeros **2 bytes del buffer** son utilizados como un **comparador de comando**, siendo el **comando 0x0064 Big Endian** el comando que redirecciona a la función **parser**.

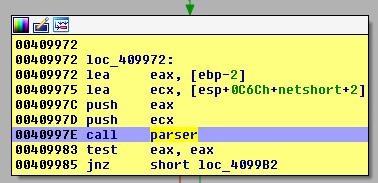
Ajustamos nuestro script nuevamente, para llegar a la función **parser** y ver si logramos triggerear el bug.



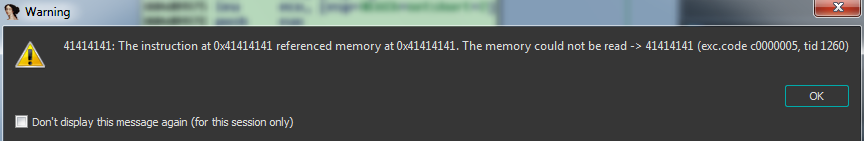
Al ejecutarlo vemos cómo logramos superar el desvío:



Y llegamos a la función **parser:**



Presionamos **F9** y vemos como el EIP se encuentra completamente controlado:



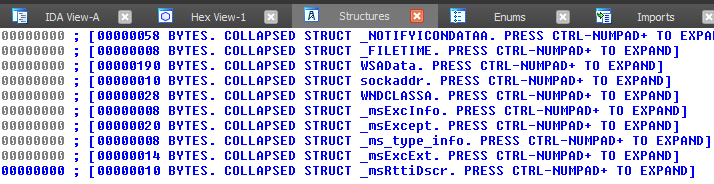
# **Explotación del bug**

Llegado este momento debemos realizar algo de reversing estático, toda información de la aplicación se mueve en base a estructuras **[ebp+offset]** por lo que es importante identificarlas.

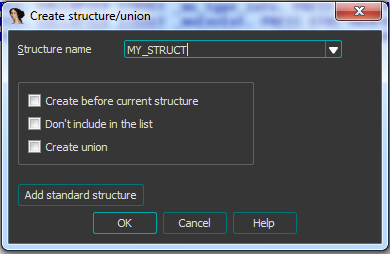
En la estructura principal podemos apreciar 2 Estructuras:

La primera offseteada desde **0x0C6C** y la segunda offseteada desde **0x0C70** por lo que vamos a crear 2 estructuras en IDA con un tamaño de 0x1000 bytes.

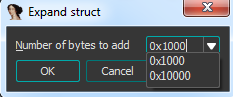
Presionamos **Shift + F9** para abrir la vista de Estructuras:



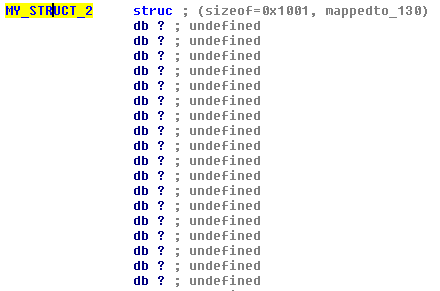
Presionamos la tecla **Insert** para crear una nueva Estructura de nombre **MY\_STRUCT**:



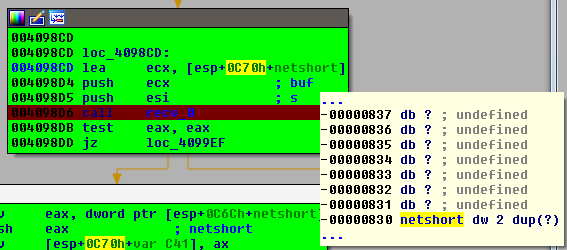
Expandimos la estructura a 0x1000 bytes:



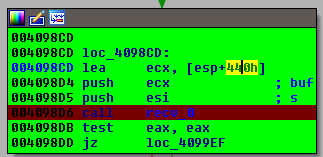
Repetimos el proceso creando una segunda estructura **MY\_STRUCT\_2** de 0x1000 bytes:



Con las estructuras declaradas vamos a comenzar a definir los atributos de cada una de ellas, comenzado por la offseteada en **0x0C70**,donde se almacena el puntero al buffer que guardará el contenido del paquete enviado cuando la función **recv\_0** sea llamada:



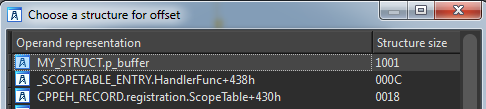
Podemos ver además que el teórico size del atributo es de 2 dword (**8 bytes**). Nos posicionamos sobre la estructura y presionamos la tecla **k** para obtener el offset correspondiente al atributo en la estructura:



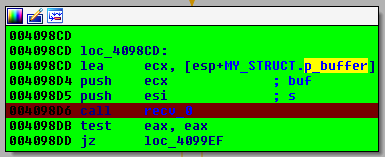
Nos dirigimos hacia el offset **440** de la primer estructura que creamos y creamos un atributo con el nombre **p\_buffer** y de tamaño **2 dword.**



Una vez hecho esto, volvemos al Basic block, nos posicionamos sobre la instrucción y presionamos la tecla **t.**

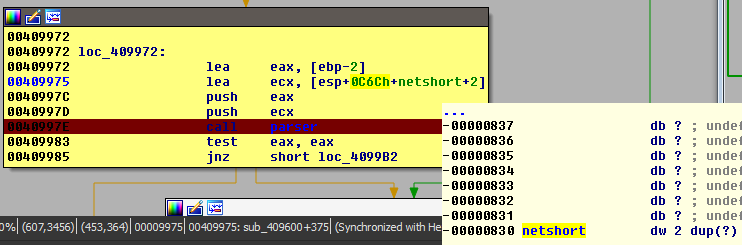


Y le asignamos el valor de nuestra struct:

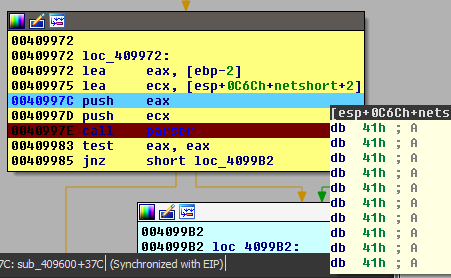


Esto nos permite volver más legible el proceso de reversing y realizar asociaciones entre las funciones.

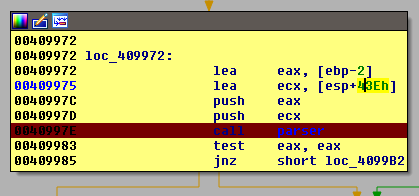
El siguiente valor importante por identificar es el puntero al buffer que se le envía como parámetro a la función **parser**:



Podemos observar cómo la función parser recibe como parámetro 2 punteros, uno de ellos apunta directamente al buffer del paquete que enviamos y es de tamaño 2 dword:



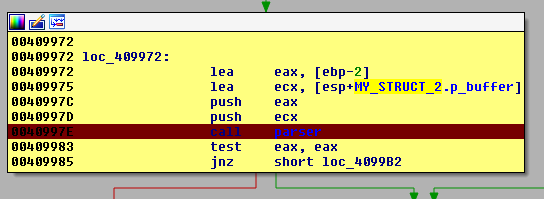
Obtenemos su offset presionando la tecla **k**.



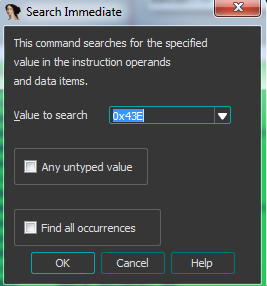
Creamos este atributo en la segunda estructura que creamos anteriormente:



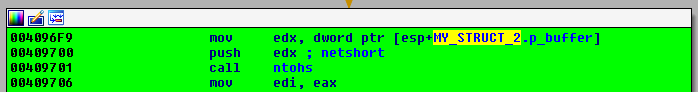
Asignamos el valor de la estructura a la instrucción del Basic block:



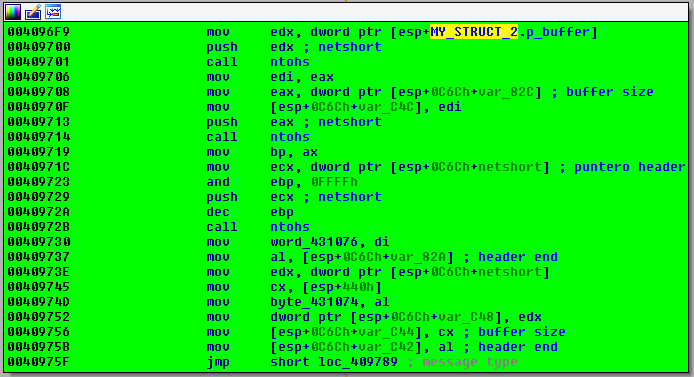
Utilizamos la búsqueda inmediata de IDA Pro para detectar todos los atributos de la estructura que offsetean de **0x43E**



Y reasignamos todos los valores que aparezcan:

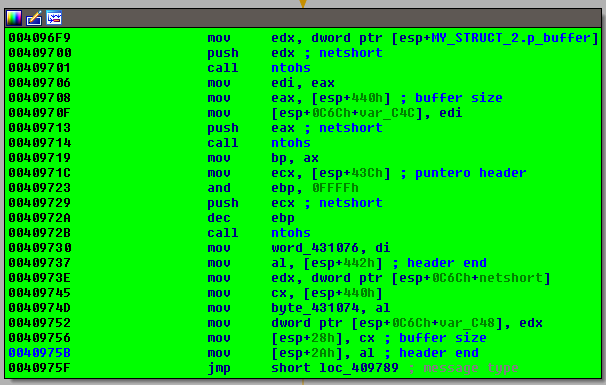


Regresamos al Basic block que procesa el header del paquete y comenzamos a descomponer los atributos de la estructura. Luego de un análisis inicial podemos detectar los siguientes atributos (señalados como comentarios):



Todos estos atributos son partes de la estructura offseteada de **0xC6C** la cual nosotros vamos a vincular con la estructura llamada **MY\_STRUCT\_2** que creamos anteriormente.

Presionamos la tecla **k** sobre todos los valores identificados para obtener sus offsets en relación con la estructura:

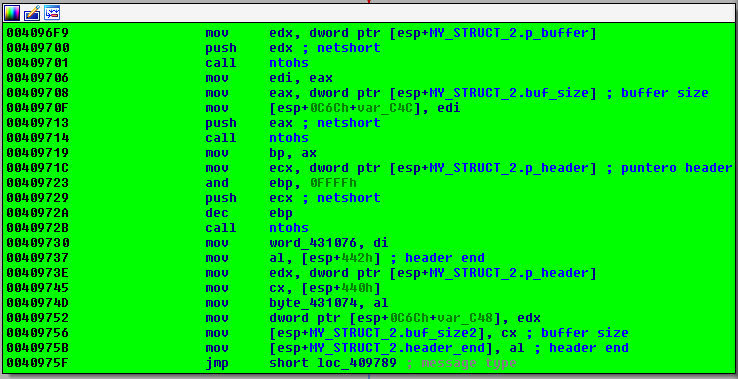


Creamos los nuevos atributos y ajustamos los existentes (si es necesario) en **MY\_STRUCT\_2**:

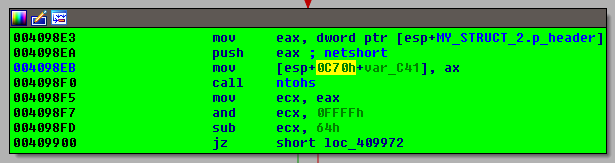




Asociamos la estructura en el Basic block presionando la tecla **t** sobre cada atributo:

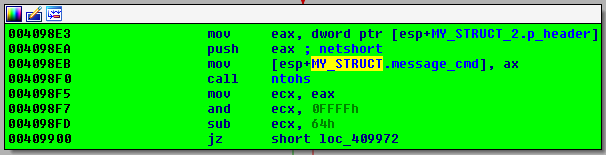


Utilizamos la búsqueda Inmediata para sustituir todos los atributos:

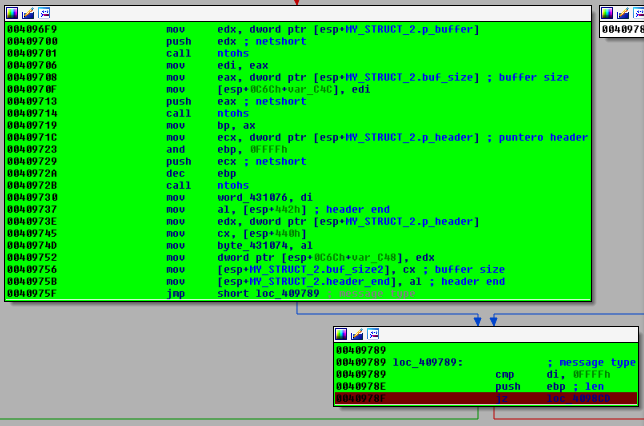


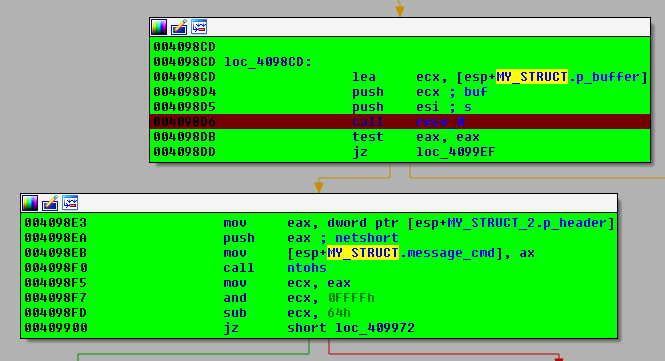
Vemos que nos falta identificar **var\_C41** que es parte de la **MY\_STRUCT** y almacena el **message\_cmd** (0x64 dword), por lo que creamos el valor en la estructura correspondiente y realizamos la asociación:

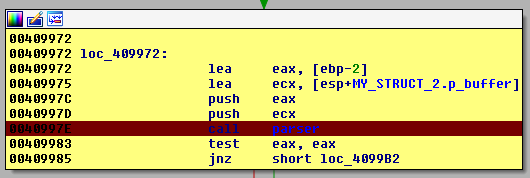




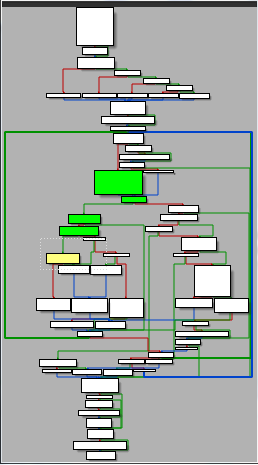
Pasando en limpio todo el proceso el Path hacia el Buffer Overflow luciría de la siguiente forma:





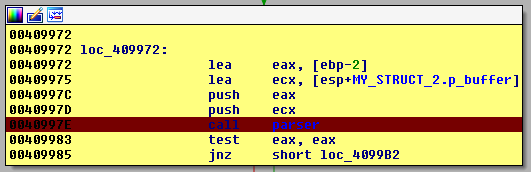


En el Graph Overview:



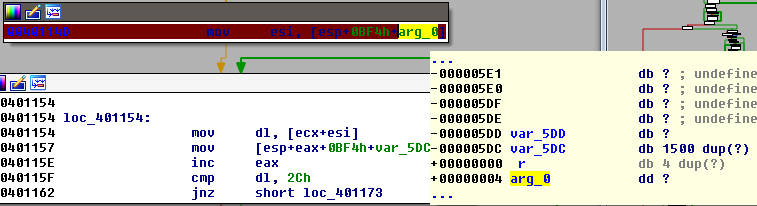
¿Mucho más legible no? Ahora queda identificar la raíz del bug, dónde se produce y por qué, dentro de la función **parser**.

De momento sabemos que la función **parser** recibe dos punteros como parámetros, uno de ellos es un puntero al buffer donde se alojan nuestras **A**, ahora debemos identificar dónde se utiliza este puntero dentro de la función **parser**.

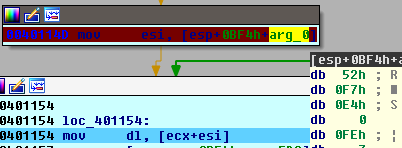


Debido al orden en que se pushean los argumentos al stack, podemos ver que **p\_buffer** será accedido como **arg\_0** dentro de la función **parser**.

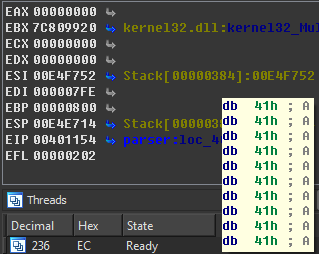
Al recorrer un poco esta función encontramos que **arg\_0** es accedido desde una estructura offseteada en **0xBF4** de la propia función:



Colocamos un break point y debuggeamos sólo para sacarnos la duda:



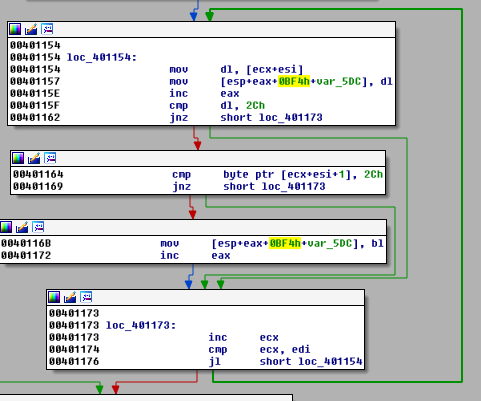
Observamos a dónde apunta este puntero:



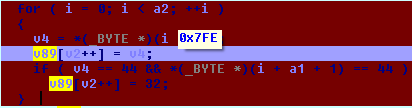


No queda duda de que es el puntero a nuestro buffer que pasamos como parámetro en el momento del llamado a la función **parser**.

Si continuamos la ejecución llegamos al siguiente bucle:



El cual comienza a copiar byte por byte el contenido de nuestro buffer (sin importar su largo) dentro de **var\_5DC** que es parte de la misma estructura que **arg\_0**. Analicemos el pseudocódigo de este loop:



Vemos cómo el size del bucle es **0x7FE** (**2046 bytes**)lo cual corresponde a **0x800 bytes** del largo de nuestro buffer menos 2 bytes del **message\_cmd** (**\x00\x64**), también podemos apreciar que el loop hace un append sobre la variable **v89** (**var\_5DC**) en cada iteración. Si analizamos el size de esta variable vemos que sólo puede contener **1500 bytes**:



Si lo vemos desde el Stack View se ve de la siguiente forma:



Con esto en mente, podemos determinar que **el bug se produce en este loop** que ira appendeando caracteres descontroladamente superando los **1500 bytes** (**debido a que controlamos el buffer size del mensaje que enviamos**) que **var\_5DC** puede contener, desbordando el buffer y sobrescribiendo **EIP** (**r**).

Creamos una nueva estructura llamada **VULN\_STRUCT** de **0x1000 bytes** (si necesitamos más la agrandaremos luego)



Identificamos el offset de **arg\_0** y creamos el atributo correspondiente con el nombre **p\_buff** con size dword:

En el Basic Block:



En la Estructura:

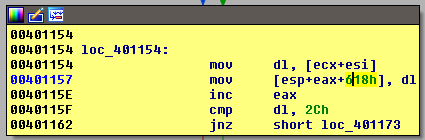


Asignamos en el Basic Block:



Realizamos el mismo proceso para la variable **var\_5DC** cuyo equivalente en la estructura será **un array char de 1500 bytes** y se llamará **vuln\_buffer.**

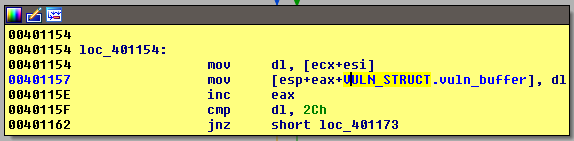
En el Basic Block:



En la Estructura:



Asignamos en el Basic Block:



Habiendo identificado el bug, comprendiendo cómo operan las estructuras del programa, la estructura del header del paquete y el tamaño del buffer vulnerable estamos listos para generar el exploit.

Para controlar **EIP** necesitaremos un buffer size mínimo de **1507 bytes**, esto es calculable debido a las siguientes necesidades:

. **vuln\_buffer** tiene un size de **1500** bytes.

. Al size que nosotros enviamos en el header del mensaje se le resta **1 byte** como regla (lo analizamos al principio de este paper) **por lo que debemos sumarle 1 byte al buffer del paquete**.

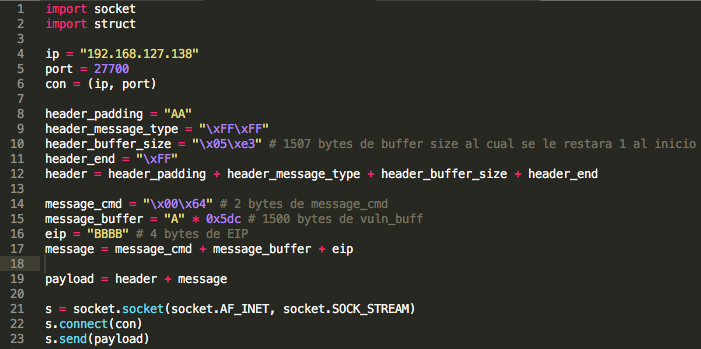
. El **message\_cmd** tiene un size de **2 bytes** que **es ignorado como buffer en el parser**, por lo que **debemos sumar 2 bytes al buffer size de nuestro paquete**.

. Con esto lograríamos llenar **vuln\_buffer** (1500 bytes) pero **debemos agregar 4 bytes mas para controlar EIP**

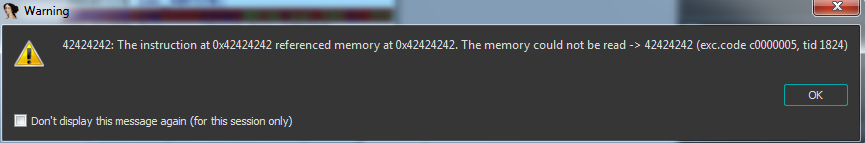
**Sumatoria = 1500 bytes vuln\_buffer + 1 byte (resta al inicio) + 2 bytes (message\_cmd) + 4 bytes (eip)**

**Sumatoria = 1507 bytes = 0x5e3 hex**

Armamos el exploit para corroborar el control sobre EIP:

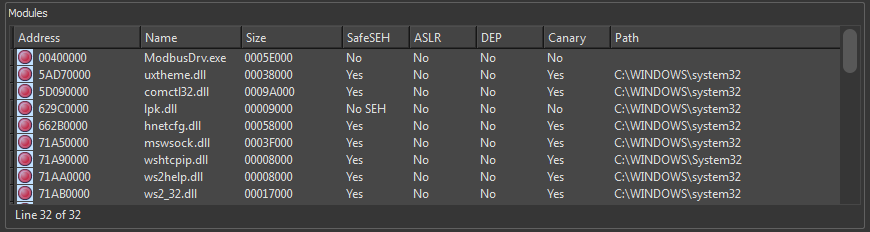


Ejecutamos el exploit y obtenemos control sobre EIP:



Como el bug deriva de un loop del cual controlamos el size, podemos determinar que **no tenemos un limite de bytes a copiar**, simplemente debemos ajustar el **buffer\_size** del paquete que enviamos para que la función **recv\_0 no falle al tratar de leer el paquete completo**.

Antes de proseguir debemos analizar las protecciones del binario para decidir si es necesario realizar ROP o no.

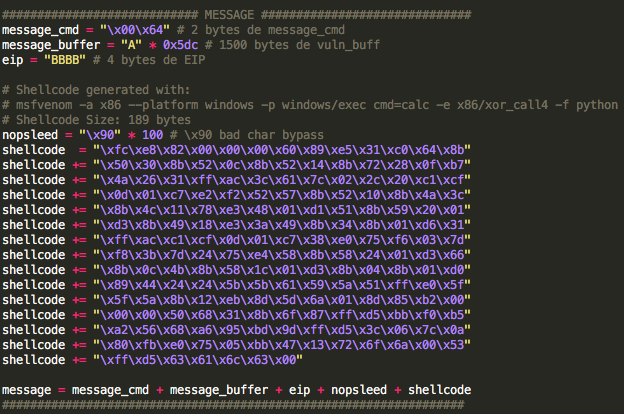


Podemos ver que no existe casi ninguna protección sobre el Binario y las librerías del Sistema que utiliza.

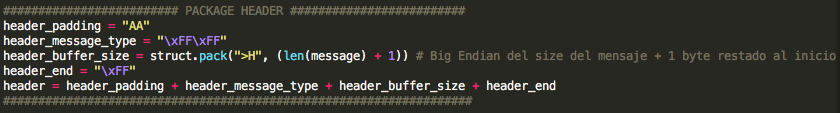
Por lo que, para lograr ejecución de código, podemos incluir nuestro shellcode junto con el nopsleed en el stack y buscar algún gadget que pase la ejecución al stack.

Como el **buffer\_size del header** depende exclusivamente del **size** del mensaje, podemos generar primero el mensaje y luego calcular el **buffer\_size del header dinámicamente**.

Para esto generamos nuestro mensaje incluyendo **nopsleed** y **shellcode**:



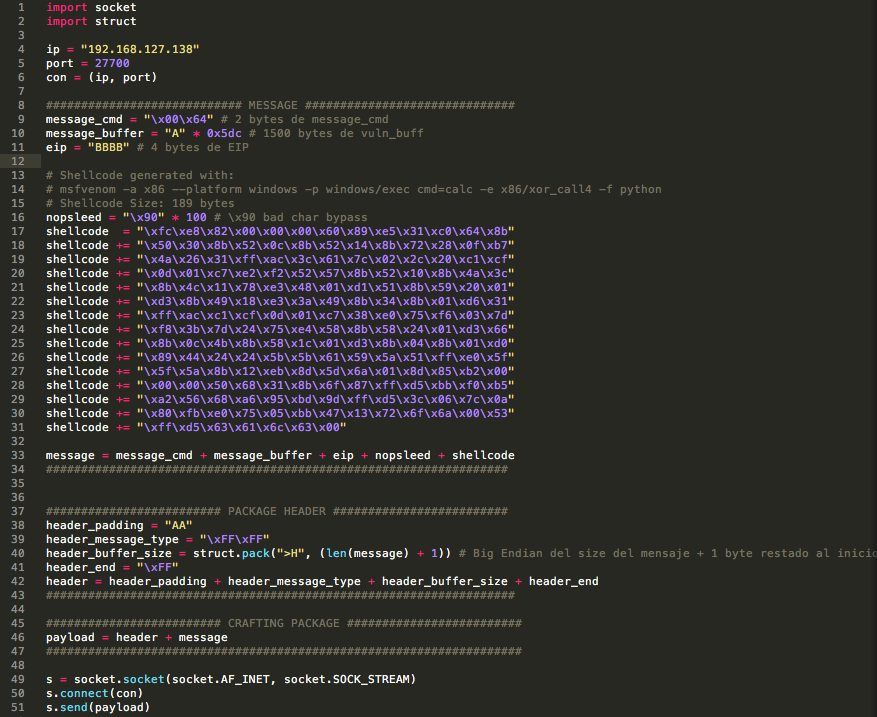
Luego armamos el **header del paquete** ajustando de forma dinámica el **buffer\_size**.

****

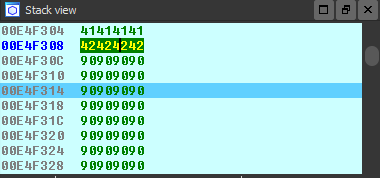
Luego de esto ensamblamos las partes del paquete:



El exploit completo:

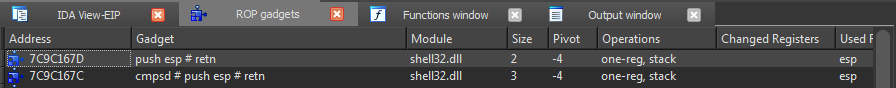


Ejecutamos el exploit hasta pisar **EIP** con **0x42424242** y analizamos el stack:



Vemos que efectivamente el **nopsleed** y el shellcode se encuentra debajo de **EIP** por lo que sólo necesitaremos un gadget para cambiar la ejecución al stack, el clásico: **push esp, ret**

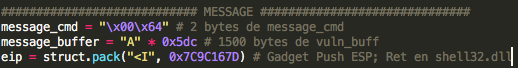
Buscamos el gadget con **IDA SPLOITER** sobre cualquier librería del sistema que no tenga **ASLR** activo:



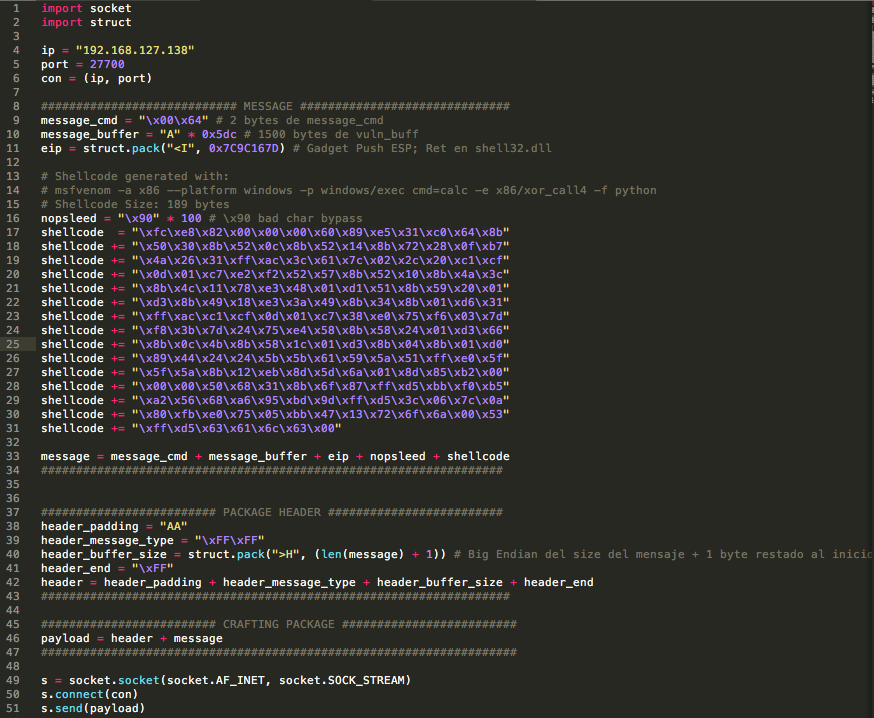
Verificamos que el gadget exista en **0x7C9C167D**:

****

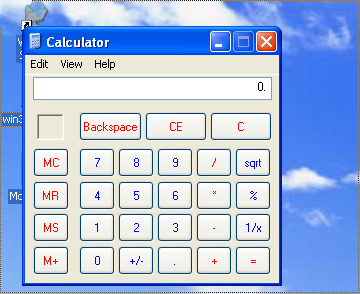
Sustituimos **EIP** por el **gadget** en nuestro exploit:



Quedando el exploit completo de la siguiente forma:



Ejecutamos el exploit y …



**Obtenemos nuestra preciada calculadora!**

# **Conclusión**

Este exploit me pareció excelente para trabajar con estructuras, si bien su dificultad no es muy grande, su análisis es muy entretenido ya que posee varias estructuras y el bug por el que se produce el overflow no es un clásico memcpy / strcpy.

¡Espero que el documento sea de utilidad para la comunidad!

